Solidity Documentation

Release 0.6.0

Ethereum

Feb 06, 2020

Contents

Solidity is an object-oriented, high-level language for implementing smart contracts. Smart contracts are programs which govern the behaviour of accounts within the Ethereum state.

Solidity was influenced by C++, Python and JavaScript and is designed to target the Ethereum Virtual Machine (EVM).

Solidity is statically typed, supports inheritance, libraries and complex user-defined types among other features.

With Solidity you can create contracts for uses such as voting, crowdfunding, blind auctions, and multi-signature wallets.

When deploying contracts, you should use the latest released version of Solidity. This is because breaking changes as well as new features and bug fixes are introduced regularly. We currently use ..a 0.x version number to indicate this fast pace of change. a 0.x version number .

> **Warning:** Solidity recently released the 0.6.x version that introduced a lot of breaking changes. Make sure you read the full list.

CHAPTER 1

Language Documentation

If you are new to the concept of smart contracts we recommend you start with *an example smart contract* written in Solidity. When you are ready for more detail, we recommend you read the *"Solidity by Example"* and *"Solidity in Depth"* sections to learn the core concepts of the language.

For further reading, try *the basics of blockchains* and details of the *Ethereum Virtual Machine*.

Hint: You can always try out code examples in your browser with the Remix IDE. Remix is a web browser based IDE that allows you to write Solidity smart contracts, then deploy and run the smart contracts. It can take a while to load, so please be patient.

Warning: As humans write software, it can have bugs. You should follow established software development best-practices when writing your smart contracts, this includes code review, testing, audits, and correctness proofs. Smart contract users are sometimes more confident with code than their authors, and blockchains and smart contracts have their own unique issues to watch out for, so before working on production code, make sure you read the security_considerations section.

If you have any questions, you can try searching for answers or asking on the Ethereum Stackexchange, or our gitter channel.

Ideas for improving Solidity or this documentation are always welcome, read our contributors guide for more details.

CHAPTER 2

Contents

Keyword Index, Search Page

2.1 Introduction to Smart Contracts

2.1.1 A Simple Smart Contract

Let us begin with a basic example that sets the value of a variable and exposes it for other contracts to access. It is fine if you do not understand everything right now, we will go into more detail later.

Storage Example

```
pragma solidity >=0.4.0 <0.7.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

The first line tells you that the source code is written for Solidity version 0.4.0, or a newer version of the language up to, but not including version 0.7.0. This is to ensure that the contract is not compilable with a new (breaking) compiler version, where it could behave differently. *Pragmas* are common instructions for compilers about how to treat the source code (e.g. pragma once).

A contract in the sense of Solidity is a collection of code (its *functions*) and data (its *state*) that resides at a specific address on the Ethereum blockchain. The line `uint storedData;` declares a state variable called `storedData` of type `uint` (*u*nsigned *int*eger of *256* bits). You can think of it as a single slot in a database that you can query and alter by calling functions of the code that manages the database. In this example, the contract defines the functions `set` and `get` that can be used to modify or retrieve the value of the variable.

To access a state variable, you do not need the prefix `this.` as is common in other languages.

This contract does not do much yet apart from (due to the infrastructure built by Ethereum) allowing anyone to store a single number that is accessible by anyone in the world without a (feasible) way to prevent you from publishing this number. Anyone could call `set` again with a different value and overwrite your number, but the number is still stored in the history of the blockchain. Later, you will see how you can impose access restrictions so that only you can alter the number.

Warning: Be careful with using Unicode text, as similar looking (or even identical) characters can have different code points and as such are encoded as a different byte array.

Note: All identifiers (contract names, function names and variable names) are restricted to the ASCII character set. It is possible to store UTF-8 encoded data in string variables.

Subcurrency Example

The following contract implements the simplest form of a cryptocurrency. The contract allows only its creator to create new coins (different issuance schemes are possible). Anyone can send coins to each other without a need for registering with a username and password, all you need is an Ethereum keypair.

```
pragma solidity >=0.5.0 <0.7.0;

contract Coin {
    // The keyword "public" makes variables
    // accessible from other contracts
    address public minter;
    mapping (address => uint) public balances;

    // Events allow clients to react to specific
    // contract changes you declare
    event Sent(address from, address to, uint amount);

    // Constructor code is only run when the contract
    // is created
    constructor() public {
        minter = msg.sender;
    }

    // Sends an amount of newly created coins to an address
    // Can only be called by the contract creator
    function mint(address receiver, uint amount) public {
        require(msg.sender == minter);
        require(amount < 1e60);
        balances[receiver] += amount;
    }

```

(continues on next page)

(continued from previous page)

```
    // Sends an amount of existing coins
    // from any caller to an address
    function send(address receiver, uint amount) public {
        require(amount <= balances[msg.sender], "Insufficient balance.");
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        emit Sent(msg.sender, receiver, amount);
    }
}
```

This contract introduces some new concepts, let us go through them one by one.

The line `address public minter;` declares a state variable of type *address*. The `address` type is a 160-bit value that does not allow any arithmetic operations. It is suitable for storing addresses of contracts, or a hash of the public half of a keypair belonging to *external accounts*.

The keyword `public` automatically generates a function that allows you to access the current value of the state variable from outside of the contract. Without this keyword, other contracts have no way to access the variable. The code of the function generated by the compiler is equivalent to the following (ignore `external` and `view` for now):

```
function minter() external view returns (address) { return minter; }
```

You could add a function like the above yourself, but you would have a function and state variable with the same name. You do not need to do this, the compiler figures it out for you.

The next line, `mapping (address => uint) public balances;` also creates a public state variable, but it is a more complex datatype. The *mapping* type maps addresses to *unsigned integers*.

Mappings can be seen as hash tables which are virtually initialised such that every possible key exists from the start and is mapped to a value whose byte-representation is all zeros. However, it is neither possible to obtain a list of all keys of a mapping, nor a list of all values. Record what you added to the mapping, or use it in a context where this is not needed. Or even better, keep a list, or use a more suitable data type.

The *getter function* created by the `public` keyword is more complex in the case of a mapping. It looks like the following:

```
function balances(address _account) external view returns (uint) {
    return balances[_account];
}
```

You can use this function to query the balance of a single account.

The line `event Sent(address from, address to, uint amount);` declares an *"event"*, which is emitted in the last line of the function `send`. Ethereum clients such as web applications can listen for these events emitted on the blockchain without much cost. As soon as it is emitted, the listener receives the arguments `from`, `to` and `amount`, which makes it possible to track transactions.

To listen for this event, you could use the following JavaScript code, which uses web3.js to create the `Coin` contract object, and any user interface calls the automatically generated `balances` function from above:

```
Coin.Sent().watch({}, '', function(error, result) {
    if (!error) {
        console.log("Coin transfer: " + result.args.amount +
            " coins were sent from " + result.args.from +
            " to " + result.args.to + ".");
        console.log("Balances now:\n" +
            "Sender: " + Coin.balances.call(result.args.from) +
            "Receiver: " + Coin.balances.call(result.args.to));
```

(continues on next page)

(continued from previous page)

```
    }
})
```

The *constructor* is a special function run during the creation of the contract and cannot be called afterwards. In this case, it permanently stores the address of the person creating the contract. The `msg` variable (together with `tx` and `block`) is a *special global variable* that contains properties which allow access to the blockchain. `msg.sender` is always the address where the current (external) function call came from.

The functions that make up the contract, and that users and contracts can call are `mint` and `send`.

The `mint` function sends an amount of newly created coins to another address. The *require* function call defines conditions that reverts all changes if not met. In this example, `require(msg.sender == minter);` ensures that only the creator of the contract can call `mint`, and `require(amount < 1e60);` ensures a maximum amount of tokens. This ensures that there are no overflow errors in the future.

The `send` function can be used by anyone (who already has some of these coins) to send coins to anyone else. If the sender does not have enough coins to send, the `require` call fails and provides the sender with an appropriate error message string.

Note: If you use this contract to send coins to an address, you will not see anything when you look at that address on a blockchain explorer, because the record that you sent coins and the changed balances are only stored in the data storage of this particular coin contract. By using events, you can create a "blockchain explorer" that tracks transactions and balances of your new coin, but you have to inspect the coin contract address and not the addresses of the coin owners.

2.1.2 Blockchain Basics

Blockchains as a concept are not too hard to understand for programmers. The reason is that most of the complications (mining, hashing, elliptic-curve cryptography, peer-to-peer networks, etc.) are just there to provide a certain set of features and promises for the platform. Once you accept these features as given, you do not have to worry about the underlying technology - or do you have to know how Amazon's AWS works internally in order to use it?

Transactions

A blockchain is a globally shared, transactional database. This means that everyone can read entries in the database just by participating in the network. If you want to change something in the database, you have to create a so-called transaction which has to be accepted by all others. The word transaction implies that the change you want to make (assume you want to change two values at the same time) is either not done at all or completely applied. Furthermore, while your transaction is being applied to the database, no other transaction can alter it.

As an example, imagine a table that lists the balances of all accounts in an electronic currency. If a transfer from one account to another is requested, the transactional nature of the database ensures that if the amount is subtracted from one account, it is always added to the other account. If due to whatever reason, adding the amount to the target account is not possible, the source account is also not modified.

Furthermore, a transaction is always cryptographically signed by the sender (creator). This makes it straightforward to guard access to specific modifications of the database. In the example of the electronic currency, a simple check ensures that only the person holding the keys to the account can transfer money from it.

Blocks

One major obstacle to overcome is what (in Bitcoin terms) is called a "double-spend attack": What happens if two transactions exist in the network that both want to empty an account? Only one of the transactions can be valid, typically the one that is accepted first. The problem is that "first" is not an objective term in a peer-to-peer network.

The abstract answer to this is that you do not have to care. A globally accepted order of the transactions will be selected for you, solving the conflict. The transactions will be bundled into what is called a "block" and then they will be executed and distributed among all participating nodes. If two transactions contradict each other, the one that ends up being second will be rejected and not become part of the block.

These blocks form a linear sequence in time and that is where the word "blockchain" derives from. Blocks are added to the chain in rather regular intervals - for Ethereum this is roughly every 17 seconds.

As part of the "order selection mechanism" (which is called "mining") it may happen that blocks are reverted from time to time, but only at the "tip" of the chain. The more blocks are added on top of a particular block, the less likely this block will be reverted. So it might be that your transactions are reverted and even removed from the blockchain, but the longer you wait, the less likely it will be.

Note: Transactions are not guaranteed to be included in the next block or any specific future block, since it is not up to the submitter of a transaction, but up to the miners to determine in which block the transaction is included.

If you want to schedule future calls of your contract, you can use the alarm clock or a similar oracle service.

2.1.3 The Ethereum Virtual Machine

Overview

The Ethereum Virtual Machine or EVM is the runtime environment for smart contracts in Ethereum. It is not only sandboxed but actually completely isolated, which means that code running inside the EVM has no access to network, filesystem or other processes. Smart contracts even have limited access to other smart contracts.

Accounts

There are two kinds of accounts in Ethereum which share the same address space: **External accounts** that are controlled by public-private key pairs (i.e. humans) and **contract accounts** which are controlled by the code stored together with the account.

The address of an external account is determined from the public key while the address of a contract is determined at the time the contract is created (it is derived from the creator address and the number of transactions sent from that address, the so-called "nonce").

Regardless of whether or not the account stores code, the two types are treated equally by the EVM.

Every account has a persistent key-value store mapping 256-bit words to 256-bit words called **storage**.

Furthermore, every account has a **balance** in Ether (in "Wei" to be exact, *1 ether* is *10**18 wei*) which can be modified by sending transactions that include Ether.

Transactions

A transaction is a message that is sent from one account to another account (which might be the same or empty, see below). It can include binary data (which is called "payload") and Ether.

If the target account contains code, that code is executed and the payload is provided as input data.

If the target account is not set (the transaction does not have a recipient or the recipient is set to `null`), the transaction creates a **new contract**. As already mentioned, the address of that contract is not the zero address but an address derived from the sender and its number of transactions sent (the "nonce"). The payload of such a contract creation transaction is taken to be EVM bytecode and executed. The output data of this execution is permanently stored as the code of the contract. This means that in order to create a contract, you do not send the actual code of the contract, but in fact code that returns that code when executed.

Note: While a contract is being created, its code is still empty. Because of that, you should not call back into the contract under construction until its constructor has finished executing.

Gas

Upon creation, each transaction is charged with a certain amount of **gas**, whose purpose is to limit the amount of work that is needed to execute the transaction and to pay for this execution at the same time. While the EVM executes the transaction, the gas is gradually depleted according to specific rules.

The **gas price** is a value set by the creator of the transaction, who has to pay `gas_price * gas` up front from the sending account. If some gas is left after the execution, it is refunded to the creator in the same way.

If the gas is used up at any point (i.e. it would be negative), an out-of-gas exception is triggered, which reverts all modifications made to the state in the current call frame.

Storage, Memory and the Stack

The Ethereum Virtual Machine has three areas where it can store data- storage, memory and the stack, which are explained in the following paragraphs.

Each account has a data area called **storage**, which is persistent between function calls and transactions. Storage is a key-value store that maps 256-bit words to 256-bit words. It is not possible to enumerate storage from within a contract, it is comparatively costly to read, and even more to initialise and modify storage. Because of this cost, you should minimize what you store in persistent storage to what the contract needs to run. Store data like derived calculations, caching, and aggregates outside of the contract. A contract can neither read nor write to any storage apart from its own.

The second data area is called **memory**, of which a contract obtains a freshly cleared instance for each message call. Memory is linear and can be addressed at byte level, but reads are limited to a width of 256 bits, while writes can be either 8 bits or 256 bits wide. Memory is expanded by a word (256-bit), when accessing (either reading or writing) a previously untouched memory word (i.e. any offset within a word). At the time of expansion, the cost in gas must be paid. Memory is more costly the larger it grows (it scales quadratically).

The EVM is not a register machine but a stack machine, so all computations are performed on a data area called the **stack**. It has a maximum size of 1024 elements and contains words of 256 bits. Access to the stack is limited to the top end in the following way: It is possible to copy one of the topmost 16 elements to the top of the stack or swap the topmost element with one of the 16 elements below it. All other operations take the topmost two (or one, or more, depending on the operation) elements from the stack and push the result onto the stack. Of course it is possible to move stack elements to storage or memory in order to get deeper access to the stack, but it is not possible to just access arbitrary elements deeper in the stack without first removing the top of the stack.

Instruction Set

The instruction set of the EVM is kept minimal in order to avoid incorrect or inconsistent implementations which could cause consensus problems. All instructions operate on the basic data type, 256-bit words or on slices of memory (or other byte arrays). The usual arithmetic, bit, logical and comparison operations are present. Conditional and unconditional jumps are possible. Furthermore, contracts can access relevant properties of the current block like its number and timestamp.

For a complete list, please see the *list of opcodes* as part of the inline assembly documentation.

Message Calls

Contracts can call other contracts or send Ether to non-contract accounts by the means of message calls. Message calls are similar to transactions, in that they have a source, a target, data payload, Ether, gas and return data. In fact, every transaction consists of a top-level message call which in turn can create further message calls.

A contract can decide how much of its remaining **gas** should be sent with the inner message call and how much it wants to retain. If an out-of-gas exception happens in the inner call (or any other exception), this will be signaled by an error value put onto the stack. In this case, only the gas sent together with the call is used up. In Solidity, the calling contract causes a manual exception by default in such situations, so that exceptions "bubble up" the call stack.

As already said, the called contract (which can be the same as the caller) will receive a freshly cleared instance of memory and has access to the call payload - which will be provided in a separate area called the **calldata**. After it has finished execution, it can return data which will be stored at a location in the caller's memory preallocated by the caller. All such calls are fully synchronous.

Calls are **limited** to a depth of 1024, which means that for more complex operations, loops should be preferred over recursive calls. Furthermore, only 63/64th of the gas can be forwarded in a message call, which causes a depth limit of a little less than 1000 in practice.

Delegatecall / Callcode and Libraries

There exists a special variant of a message call, named **delegatecall** which is identical to a message call apart from the fact that the code at the target address is executed in the context of the calling contract and `msg.sender` and `msg.value` do not change their values.

This means that a contract can dynamically load code from a different address at runtime. Storage, current address and balance still refer to the calling contract, only the code is taken from the called address.

This makes it possible to implement the "library" feature in Solidity: Reusable library code that can be applied to a contract's storage, e.g. in order to implement a complex data structure.

Logs

It is possible to store data in a specially indexed data structure that maps all the way up to the block level. This feature called **logs** is used by Solidity in order to implement *events*. Contracts cannot access log data after it has been created, but they can be efficiently accessed from outside the blockchain. Since some part of the log data is stored in bloom filters, it is possible to search for this data in an efficient and cryptographically secure way, so network peers that do not download the whole blockchain (so-called "light clients") can still find these logs.

Create

Contracts can even create other contracts using a special opcode (i.e. they do not simply call the zero address as a transaction would). The only difference between these **create calls** and normal message calls is that the payload data is executed and the result stored as code and the caller / creator receives the address of the new contract on the stack.

Deactivate and Self-destruct

The only way to remove code from the blockchain is when a contract at that address performs the `selfdestruct` operation. The remaining Ether stored at that address is sent to a designated target and then the storage and code is removed from the state. Removing the contract in theory sounds like a good idea, but it is potentially dangerous, as if someone sends Ether to removed contracts, the Ether is forever lost.

Warning: Even if a contract is removed by "selfdestruct", it is still part of the history of the blockchain and probably retained by most Ethereum nodes. So using "selfdestruct" is not the same as deleting data from a hard disk.

Note: Even if a contract's code does not contain a call to `selfdestruct`, it can still perform that operation using `delegatecall` or `callcode`.

If you want to deactivate your contracts, you should instead **disable** them by changing some internal state which causes all functions to revert. This makes it impossible to use the contract, as it returns Ether immediately.

2.2 Installing the Solidity Compiler

2.2.1 Versioning

Solidity versions follow semantic versioning and in addition to releases, **nightly development builds** are also made available. The nightly builds are not guaranteed to be working and despite best efforts they might contain undocumented and/or broken changes. We recommend using the latest release. Package installers below will use the latest release.

2.2.2 Docker

Docker images of Solidity builds are available using the `solc` image from the `ethereum` organisation. Use the `stable` tag for the latest released version, and `nightly` for potentially unstable changes in the develop branch.

The Docker image runs the compiler executable, so you can pass all compiler arguments to it. For example, the command below pulls the stable version of the `solc` image (if you do not have it already), and runs it in a new container, passing the `--help` argument.

```
docker run ethereum/solc:stable --help
```

You can also specify release build versions in the tag, for example, for the 0.5.4 release.

```
docker run ethereum/solc:0.5.4 --help
```

To use the Docker image to compile Solidity files on the host machine mount a local folder for input and output, and specify the contract to compile. For example.

```
docker run -v /local/path:/sources ethereum/solc:stable -o /sources/output --abi --
↪bin /sources/Contract.sol
```

You can also use the standard JSON interface (which is recommended when using the compiler with tooling). When using this interface it is not necessary to mount any directories.

```
docker run ethereum/solc:stable --standard-json < input.json > output.json
```

2.2.3 Binary Packages

Binary packages of Solidity are available at solidity/releases.

We also have PPAs for Ubuntu, you can get the latest stable version using the following commands:

```
sudo add-apt-repository ppa:ethereum/ethereum
sudo apt-get update
sudo apt-get install solc
```

The nightly version can be installed using these commands:

```
sudo add-apt-repository ppa:ethereum/ethereum
sudo add-apt-repository ppa:ethereum/ethereum-dev
sudo apt-get update
sudo apt-get install solc
```

We are also releasing a snap package, which is installable in all the supported Linux distros. To install the latest stable version of solc:

```
sudo snap install solc
```

If you want to help testing the latest development version of Solidity with the most recent changes, please use the following:

```
sudo snap install solc --edge
```

Note: The `solc` snap uses strict confinement. This is the most secure mode for snap packages but it comes with limitations, like accessing only the files in your `/home` and `/media` directories. For more information, go to Demystifying Snap Confinement.

Arch Linux also has packages, albeit limited to the latest development version:

```
pacman -S solidity
```

We distribute the Solidity compiler through Homebrew as a build-from-source version. Pre-built bottles are currently not supported.

```
brew update
brew upgrade
brew tap ethereum/ethereum
brew install solidity
```

To install the most recent 0.4.x / 0.5.x version of Solidity you can also use `brew install solidity@4` and `brew install solidity@5`, respectively.

If you need a specific version of Solidity you can install a Homebrew formula directly from Github.

View solidity.rb commits on Github.

Follow the history links until you have a raw file link of a specific commit of `solidity.rb`.

Install it using `brew`:

```
brew unlink solidity
# eg. Install 0.4.8
brew install https://raw.githubusercontent.com/ethereum/homebrew-ethereum/
↪77cce03da9f289e5a3ffe579840d3c5dc0a62717/solidity.rb
```

Gentoo Linux also provides a solidity package that can be installed using `emerge`:

```
emerge dev-lang/solidity
```

2.2.4 Building from Source

Prerequisites - All Operating Systems

The following are dependencies for all builds of Solidity:

Software	Notes
CMake (version 3.9+)	Cross-platform build file generator.
Boost (version 1.65+)	C++ libraries.
Git	Command-line tool for retrieving source code.
z3 (version 4.6+, Optional)	For use with SMT checker.
cvc4 (Optional)	For use with SMT checker.

Note: Solidity versions prior to 0.5.10 can fail to correctly link against Boost versions 1.70+. A possible workaround is to temporarily rename `<Boost install path>/lib/cmake/Boost-1.70.0` prior to running the cmake command to configure solidity.

Starting from 0.5.10 linking against Boost 1.70+ should work without manual intervention.

Minimum compiler versions

The following C++ compilers and their minimum versions can build the Solidity codebase:

- GCC, version 5+
- Clang, version 3.4+
- MSVC, version 2017+

Prerequisites - macOS

For macOS builds, ensure that you have the latest version of Xcode installed. This contains the Clang C++ compiler, the Xcode IDE and other Apple development tools which are required for building C++ applications on OS X. If you are installing Xcode for the first time, or have just installed a new version then you will need to agree to the license before you can do command-line builds:

```
sudo xcodebuild -license accept
```

Our OS X build script uses the Homebrew package manager for installing external dependencies. Here's how to uninstall Homebrew, if you ever want to start again from scratch.

Prerequisites - Windows

You need to install the following dependencies for Windows builds of Solidity:

Software	Notes
Visual Studio 2017 Build Tools	C++ compiler
Visual Studio 2017 (Optional)	C++ compiler and dev environment.

If you already have one IDE and only need the compiler and libraries, you could install Visual Studio 2017 Build Tools.

Visual Studio 2017 provides both IDE and necessary compiler and libraries. So if you have not got an IDE and prefer to develop solidity, Visual Studio 2017 may be a choice for you to get everything setup easily.

Here is the list of components that should be installed in Visual Studio 2017 Build Tools or Visual Studio 2017:

- Visual Studio C++ core features
- VC++ 2017 v141 toolset (x86,x64)
- Windows Universal CRT SDK
- Windows 8.1 SDK
- C++/CLI support

Dependencies Helper Script

We have a helper script which you can use to install all required external dependencies on macOS, Windows and on numerous Linux distros.

```
./scripts/install_deps.sh
```

Or, on Windows:

```
scripts\install_deps.bat
```

Clone the Repository

To clone the source code, execute the following command:

```
git clone --recursive https://github.com/ethereum/solidity.git
cd solidity
```

If you want to help developing Solidity, you should fork Solidity and add your personal fork as a second remote:

```
git remote add personal git@github.com:[username]/solidity.git
```

Command-Line Build

Be sure to install External Dependencies (see above) before build.

Solidity project uses CMake to configure the build. You might want to install ccache to speed up repeated builds. CMake will pick it up automatically. Building Solidity is quite similar on Linux, macOS and other Unices:

```
mkdir build
cd build
cmake .. && make
```

or even easier on Linux and macOS, you can run:

```
#note: this will install binaries solc and soltest at usr/local/bin
./scripts/build.sh
```

Warning: BSD builds should work, but are untested by the Solidity team.

And for Windows:

```
mkdir build
cd build
cmake -G "Visual Studio 15 2017 Win64" ..
```

This latter set of instructions should result in the creation of **solidity.sln** in that build directory. Double-clicking on that file should result in Visual Studio firing up. We suggest building **Release** configuration, but all others work.

Alternatively, you can build for Windows on the command-line, like so:

```
cmake --build . --config Release
```

2.2.5 CMake options

If you are interested what CMake options are available run `cmake .. -LH`.

SMT Solvers

Solidity can be built against SMT solvers and will do so by default if they are found in the system. Each solver can be disabled by a *cmake* option.

Note: In some cases, this can also be a potential workaround for build failures.

Inside the build folder you can disable them, since they are enabled by default:

```
# disables only Z3 SMT Solver.
cmake .. -DUSE_Z3=OFF

# disables only CVC4 SMT Solver.
cmake .. -DUSE_CVC4=OFF

# disables both Z3 and CVC4
cmake .. -DUSE_CVC4=OFF -DUSE_Z3=OFF
```

2.2.6 The version string in detail

The Solidity version string contains four parts:

- the version number
- pre-release tag, usually set to `develop.YYYY.MM.DD` or `nightly.YYYY.MM.DD`
- commit in the format of `commit.GITHASH`
- platform, which has an arbitrary number of items, containing details about the platform and compiler

If there are local modifications, the commit will be postfixed with `.mod`.

These parts are combined as required by Semver, where the Solidity pre-release tag equals to the Semver pre-release and the Solidity commit and platform combined make up the Semver build metadata.

A release example: `0.4.8+commit.60cc1668.Emscripten.clang`.

A pre-release example: `0.4.9-nightly.2017.1.17+commit.6ecb4aa3.Emscripten.clang`

2.2.7 Important information about versioning

After a release is made, the patch version level is bumped, because we assume that only patch level changes follow. When changes are merged, the version should be bumped according to semver and the severity of the change. Finally, a release is always made with the version of the current nightly build, but without the `prerelease` specifier.

Example:

0. the 0.4.0 release is made
1. nightly build has a version of 0.4.1 from now on
2. non-breaking changes are introduced - no change in version
3. a breaking change is introduced - version is bumped to 0.5.0
4. the 0.5.0 release is made

This behaviour works well with the *version pragma*.

2.3 Solidity by Example

2.3.1 Voting

The following contract is quite complex, but showcases a lot of Solidity's features. It implements a voting contract. Of course, the main problems of electronic voting is how to assign voting rights to the correct persons and how to prevent manipulation. We will not solve all problems here, but at least we will show how delegated voting can be done so that vote counting is **automatic and completely transparent** at the same time.

The idea is to create one contract per ballot, providing a short name for each option. Then the creator of the contract who serves as chairperson will give the right to vote to each address individually.

The persons behind the addresses can then choose to either vote themselves or to delegate their vote to a person they trust.

At the end of the voting time, `winningProposal()` will return the proposal with the largest number of votes.

```
pragma solidity >=0.4.22 <0.7.0;

/// @title Voting with delegation.
contract Ballot {
    // This declares a new complex type which will
    // be used for variables later.
    // It will represent a single voter.
    struct Voter {
        uint weight; // weight is accumulated by delegation
        bool voted;  // if true, that person already voted
        address delegate; // person delegated to
        uint vote;   // index of the voted proposal
```

(continues on next page)

(continued from previous page)

```
    }

    // This is a type for a single proposal.
    struct Proposal {
        bytes32 name;   // short name (up to 32 bytes)
        uint voteCount; // number of accumulated votes
    }

    address public chairperson;

    // This declares a state variable that
    // stores a `Voter` struct for each possible address.
    mapping(address => Voter) public voters;

    // A dynamically-sized array of `Proposal` structs.
    Proposal[] public proposals;

    /// Create a new ballot to choose one of `proposalNames`.
    constructor(bytes32[] memory proposalNames) public {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;

        // For each of the provided proposal names,
        // create a new proposal object and add it
        // to the end of the array.
        for (uint i = 0; i < proposalNames.length; i++) {
            // `Proposal({...})` creates a temporary
            // Proposal object and `proposals.push(...)`
            // appends it to the end of `proposals`.
            proposals.push(Proposal({
                name: proposalNames[i],
                voteCount: 0
            }));
        }
    }

    // Give `voter` the right to vote on this ballot.
    // May only be called by `chairperson`.
    function giveRightToVote(address voter) public {
        // If the first argument of `require` evaluates
        // to `false`, execution terminates and all
        // changes to the state and to Ether balances
        // are reverted.
        // This used to consume all gas in old EVM versions, but
        // not anymore.
        // It is often a good idea to use `require` to check if
        // functions are called correctly.
        // As a second argument, you can also provide an
        // explanation about what went wrong.
        require(
            msg.sender == chairperson,
            "Only chairperson can give right to vote."
        );
        require(
            !voters[voter].voted,
            "The voter already voted."
        );
```

(continues on next page)

(continued from previous page)

```
        require(voters[voter].weight == 0);
        voters[voter].weight = 1;
    }

    /// Delegate your vote to the voter `to`.
    function delegate(address to) public {
        // assigns reference
        Voter storage sender = voters[msg.sender];
        require(!sender.voted, "You already voted.");

        require(to != msg.sender, "Self-delegation is disallowed.");

        // Forward the delegation as long as
        // `to` also delegated.
        // In general, such loops are very dangerous,
        // because if they run too long, they might
        // need more gas than is available in a block.
        // In this case, the delegation will not be executed,
        // but in other situations, such loops might
        // cause a contract to get "stuck" completely.
        while (voters[to].delegate != address(0)) {
            to = voters[to].delegate;

            // We found a loop in the delegation, not allowed.
            require(to != msg.sender, "Found loop in delegation.");
        }

        // Since `sender` is a reference, this
        // modifies `voters[msg.sender].voted`
        sender.voted = true;
        sender.delegate = to;
        Voter storage delegate_ = voters[to];
        if (delegate_.voted) {
            // If the delegate already voted,
            // directly add to the number of votes
            proposals[delegate_.vote].voteCount += sender.weight;
        } else {
            // If the delegate did not vote yet,
            // add to her weight.
            delegate_.weight += sender.weight;
        }
    }

    /// Give your vote (including votes delegated to you)
    /// to proposal `proposals[proposal].name`.
    function vote(uint proposal) public {
        Voter storage sender = voters[msg.sender];
        require(sender.weight != 0, "Has no right to vote");
        require(!sender.voted, "Already voted.");
        sender.voted = true;
        sender.vote = proposal;

        // If `proposal` is out of the range of the array,
        // this will throw automatically and revert all
        // changes.
        proposals[proposal].voteCount += sender.weight;
    }
```

(continues on next page)

(continued from previous page)

```
    /// @dev Computes the winning proposal taking all
    /// previous votes into account.
    function winningProposal() public view
            returns (uint winningProposal_)
    {
        uint winningVoteCount = 0;
        for (uint p = 0; p < proposals.length; p++) {
            if (proposals[p].voteCount > winningVoteCount) {
                winningVoteCount = proposals[p].voteCount;
                winningProposal_ = p;
            }
        }
    }

    // Calls winningProposal() function to get the index
    // of the winner contained in the proposals array and then
    // returns the name of the winner
    function winnerName() public view
            returns (bytes32 winnerName_)
    {
        winnerName_ = proposals[winningProposal()].name;
    }
}
```

Possible Improvements

Currently, many transactions are needed to assign the rights to vote to all participants. Can you think of a better way?

2.3.2 Blind Auction

In this section, we will show how easy it is to create a completely blind auction contract on Ethereum. We will start with an open auction where everyone can see the bids that are made and then extend this contract into a blind auction where it is not possible to see the actual bid until the bidding period ends.

Simple Open Auction

The general idea of the following simple auction contract is that everyone can send their bids during a bidding period. The bids already include sending money / Ether in order to bind the bidders to their bid. If the highest bid is raised, the previously highest bidder gets their money back. After the end of the bidding period, the contract has to be called manually for the beneficiary to receive their money - contracts cannot activate themselves.

```
pragma solidity >=0.4.22 <0.7.0;

contract SimpleAuction {
    // Parameters of the auction. Times are either
    // absolute unix timestamps (seconds since 1970-01-01)
    // or time periods in seconds.
    address payable public beneficiary;
    uint public auctionEndTime;

    // Current state of the auction.
```

(continues on next page)

(continued from previous page)

```
    address public highestBidder;
    uint public highestBid;

    // Allowed withdrawals of previous bids
    mapping(address => uint) pendingReturns;

    // Set to true at the end, disallows any change.
    // By default initialized to `false`.
    bool ended;

    // Events that will be emitted on changes.
    event HighestBidIncreased(address bidder, uint amount);
    event AuctionEnded(address winner, uint amount);

    // The following is a so-called natspec comment,
    // recognizable by the three slashes.
    // It will be shown when the user is asked to
    // confirm a transaction.

    /// Create a simple auction with `_biddingTime`
    /// seconds bidding time on behalf of the
    /// beneficiary address `_beneficiary`.
    constructor(
        uint _biddingTime,
        address payable _beneficiary
    ) public {
        beneficiary = _beneficiary;
        auctionEndTime = now + _biddingTime;
    }

    /// Bid on the auction with the value sent
    /// together with this transaction.
    /// The value will only be refunded if the
    /// auction is not won.
    function bid() public payable {
        // No arguments are necessary, all
        // information is already part of
        // the transaction. The keyword payable
        // is required for the function to
        // be able to receive Ether.

        // Revert the call if the bidding
        // period is over.
        require(
            now <= auctionEndTime,
            "Auction already ended."
        );

        // If the bid is not higher, send the
        // money back (the failing require
        // will revert all changes in this
        // function execution including
        // it having received the money).
        require(
            msg.value > highestBid,
            "There already is a higher bid."
        );
```

(continues on next page)

(continued from previous page)

```
        if (highestBid != 0) {
            // Sending back the money by simply using
            // highestBidder.send(highestBid) is a security risk
            // because it could execute an untrusted contract.
            // It is always safer to let the recipients
            // withdraw their money themselves.
            pendingReturns[highestBidder] += highestBid;
        }
        highestBidder = msg.sender;
        highestBid = msg.value;
        emit HighestBidIncreased(msg.sender, msg.value);
    }

    /// Withdraw a bid that was overbid.
    function withdraw() public returns (bool) {
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            // It is important to set this to zero because the recipient
            // can call this function again as part of the receiving call
            // before `send` returns.
            pendingReturns[msg.sender] = 0;

            if (!msg.sender.send(amount)) {
                // No need to call throw here, just reset the amount owing
                pendingReturns[msg.sender] = amount;
                return false;
            }
        }
        return true;
    }

    /// End the auction and send the highest bid
    /// to the beneficiary.
    function auctionEnd() public {
        // It is a good guideline to structure functions that interact
        // with other contracts (i.e. they call functions or send Ether)
        // into three phases:
        // 1. checking conditions
        // 2. performing actions (potentially changing conditions)
        // 3. interacting with other contracts
        // If these phases are mixed up, the other contract could call
        // back into the current contract and modify the state or cause
        // effects (ether payout) to be performed multiple times.
        // If functions called internally include interaction with external
        // contracts, they also have to be considered interaction with
        // external contracts.

        // 1. Conditions
        require(now >= auctionEndTime, "Auction not yet ended.");
        require(!ended, "auctionEnd has already been called.");

        // 2. Effects
        ended = true;
        emit AuctionEnded(highestBidder, highestBid);

        // 3. Interaction
```

(continues on next page)

(continued from previous page)

```
        beneficiary.transfer(highestBid);
    }
}
```

Blind Auction

The previous open auction is extended to a blind auction in the following. The advantage of a blind auction is that there is no time pressure towards the end of the bidding period. Creating a blind auction on a transparent computing platform might sound like a contradiction, but cryptography comes to the rescue.

During the **bidding period**, a bidder does not actually send their bid, but only a hashed version of it. Since it is currently considered practically impossible to find two (sufficiently long) values whose hash values are equal, the bidder commits to the bid by that. After the end of the bidding period, the bidders have to reveal their bids: They send their values unencrypted and the contract checks that the hash value is the same as the one provided during the bidding period.

Another challenge is how to make the auction **binding and blind** at the same time: The only way to prevent the bidder from just not sending the money after they won the auction is to make them send it together with the bid. Since value transfers cannot be blinded in Ethereum, anyone can see the value.

The following contract solves this problem by accepting any value that is larger than the highest bid. Since this can of course only be checked during the reveal phase, some bids might be **invalid**, and this is on purpose (it even provides an explicit flag to place invalid bids with high value transfers): Bidders can confuse competition by placing several high or low invalid bids.

```
pragma solidity >0.4.23 <0.7.0;

contract BlindAuction {
    struct Bid {
        bytes32 blindedBid;
        uint deposit;
    }

    address payable public beneficiary;
    uint public biddingEnd;
    uint public revealEnd;
    bool public ended;

    mapping(address => Bid[]) public bids;

    address public highestBidder;
    uint public highestBid;

    // Allowed withdrawals of previous bids
    mapping(address => uint) pendingReturns;

    event AuctionEnded(address winner, uint highestBid);

    /// Modifiers are a convenient way to validate inputs to
    /// functions. `onlyBefore` is applied to `bid` below:
    /// The new function body is the modifier's body where
    /// `_` is replaced by the old function body.
    modifier onlyBefore(uint _time) { require(now < _time); _; }
    modifier onlyAfter(uint _time) { require(now > _time); _; }
```

(continues on next page)

(continued from previous page)

```
    constructor(
        uint _biddingTime,
        uint _revealTime,
        address payable _beneficiary
    ) public {
        beneficiary = _beneficiary;
        biddingEnd = now + _biddingTime;
        revealEnd = biddingEnd + _revealTime;
    }

    /// Place a blinded bid with `_blindedBid` =
    /// keccak256(abi.encodePacked(value, fake, secret)).
    /// The sent ether is only refunded if the bid is correctly
    /// revealed in the revealing phase. The bid is valid if the
    /// ether sent together with the bid is at least "value" and
    /// "fake" is not true. Setting "fake" to true and sending
    /// not the exact amount are ways to hide the real bid but
    /// still make the required deposit. The same address can
    /// place multiple bids.
    function bid(bytes32 _blindedBid)
        public
        payable
        onlyBefore(biddingEnd)
    {
        bids[msg.sender].push(Bid({
            blindedBid: _blindedBid,
            deposit: msg.value
        }));
    }

    /// Reveal your blinded bids. You will get a refund for all
    /// correctly blinded invalid bids and for all bids except for
    /// the totally highest.
    function reveal(
        uint[] memory _values,
        bool[] memory _fake,
        bytes32[] memory _secret
    )
        public
        onlyAfter(biddingEnd)
        onlyBefore(revealEnd)
    {
        uint length = bids[msg.sender].length;
        require(_values.length == length);
        require(_fake.length == length);
        require(_secret.length == length);

        uint refund;
        for (uint i = 0; i < length; i++) {
            Bid storage bidToCheck = bids[msg.sender][i];
            (uint value, bool fake, bytes32 secret) =
                    (_values[i], _fake[i], _secret[i]);
            if (bidToCheck.blindedBid != keccak256(abi.encodePacked(value, fake,␣
↪secret))) {
                // Bid was not actually revealed.
                // Do not refund deposit.
                continue;
```

(continues on next page)

(continued from previous page)

```
            }
            refund += bidToCheck.deposit;
            if (!fake && bidToCheck.deposit >= value) {
                if (placeBid(msg.sender, value))
                    refund -= value;
            }
            // Make it impossible for the sender to re-claim
            // the same deposit.
            bidToCheck.blindedBid = bytes32(0);
        }
        msg.sender.transfer(refund);
    }

    /// Withdraw a bid that was overbid.
    function withdraw() public {
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            // It is important to set this to zero because the recipient
            // can call this function again as part of the receiving call
            // before `transfer` returns (see the remark above about
            // conditions -> effects -> interaction).
            pendingReturns[msg.sender] = 0;

            msg.sender.transfer(amount);
        }
    }

    /// End the auction and send the highest bid
    /// to the beneficiary.
    function auctionEnd()
        public
        onlyAfter(revealEnd)
    {
        require(!ended);
        emit AuctionEnded(highestBidder, highestBid);
        ended = true;
        beneficiary.transfer(highestBid);
    }

    // This is an "internal" function which means that it
    // can only be called from the contract itself (or from
    // derived contracts).
    function placeBid(address bidder, uint value) internal
            returns (bool success)
    {
        if (value <= highestBid) {
            return false;
        }
        if (highestBidder != address(0)) {
            // Refund the previously highest bidder.
            pendingReturns[highestBidder] += highestBid;
        }
        highestBid = value;
        highestBidder = bidder;
        return true;
    }
}
```

2.3.3 Safe Remote Purchase

Purchasing goods remotely currently requires multiple parties that need to trust each other. The simplest configuration involves a seller and a buyer. The buyer would like to receive an item from the seller and the seller would like to get money (or an equivalent) in return. The problematic part is the shipment here: There is no way to determine for sure that the item arrived at the buyer.

There are multiple ways to solve this problem, but all fall short in one or the other way. In the following example, both parties have to put twice the value of the item into the contract as escrow. As soon as this happened, the money will stay locked inside the contract until the buyer confirms that they received the item. After that, the buyer is returned the value (half of their deposit) and the seller gets three times the value (their deposit plus the value). The idea behind this is that both parties have an incentive to resolve the situation or otherwise their money is locked forever.

This contract of course does not solve the problem, but gives an overview of how you can use state machine-like constructs inside a contract.

```
pragma solidity >=0.4.22 <0.7.0;

contract Purchase {
    uint public value;
    address payable public seller;
    address payable public buyer;

    enum State { Created, Locked, Release, Inactive }
    // The state variable has a default value of the first member, `State.created`
    State public state;

    modifier condition(bool _condition) {
        require(_condition);
        _;
    }

    modifier onlyBuyer() {
        require(
            msg.sender == buyer,
            "Only buyer can call this."
        );
        _;
    }

    modifier onlySeller() {
        require(
            msg.sender == seller,
            "Only seller can call this."
        );
        _;
    }

    modifier inState(State _state) {
        require(
            state == _state,
            "Invalid state."
        );
        _;
    }

    event Aborted();
    event PurchaseConfirmed();
```

(continues on next page)

(continued from previous page)

```
    event ItemReceived();
    event SellerRefunded();

    // Ensure that `msg.value` is an even number.
    // Division will truncate if it is an odd number.
    // Check via multiplication that it wasn't an odd number.
    constructor() public payable {
        seller = msg.sender;
        value = msg.value / 2;
        require((2 * value) == msg.value, "Value has to be even.");
    }

    /// Abort the purchase and reclaim the ether.
    /// Can only be called by the seller before
    /// the contract is locked.
    function abort()
        public
        onlySeller
        inState(State.Created)
    {
        emit Aborted();
        state = State.Inactive;
        // We use transfer here directly. It is
        // reentrancy-safe, because it is the
        // last call in this function and we
        // already changed the state.
        seller.transfer(address(this).balance);
    }

    /// Confirm the purchase as buyer.
    /// Transaction has to include `2 * value` ether.
    /// The ether will be locked until confirmReceived
    /// is called.
    function confirmPurchase()
        public
        inState(State.Created)
        condition(msg.value == (2 * value))
        payable
    {
        emit PurchaseConfirmed();
        buyer = msg.sender;
        state = State.Locked;
    }

    /// Confirm that you (the buyer) received the item.
    /// This will release the locked ether.
    function confirmReceived()
        public
        onlyBuyer
        inState(State.Locked)
    {
        emit ItemReceived();
        // It is important to change the state first because
        // otherwise, the contracts called using `send` below
        // can call in again here.
        state = State.Release;
```

(continues on next page)

(continued from previous page)

```
        buyer.transfer(value);
    }

    /// This function refunds the seller, i.e.
    /// pays back the locked funds of the seller.
    function refundSeller()
        public
        onlySeller
        inState(State.Release)
    {
        emit SellerRefunded();
        // It is important to change the state first because
        // otherwise, the contracts called using `send` below
        // can call in again here.
        state = State.Inactive;

        seller.transfer(3 * value);
    }
}
```

2.3.4 Micropayment Channel

In this section we will learn how to build an example implementation of a payment channel. It uses cryptographic signatures to make repeated transfers of Ether between the same parties secure, instantaneous, and without transaction fees. For the example, we need to understand how to sign and verify signatures, and setup the payment channel.

Creating and verifying signatures

Imagine Alice wants to send a quantity of Ether to Bob, i.e. Alice is the sender and the Bob is the recipient.

Alice only needs to send cryptographically signed messages off-chain (e.g. via email) to Bob and it is similar to writing checks.

Alice and Bob use signatures to authorise transactions, which is possible with smart contracts on Ethereum. Alice will build a simple smart contract that lets her transmit Ether, but instead of calling a function herself to initiate a payment, she will let Bob do that, and therefore pay the transaction fee.

The contract will work as follows:

1. Alice deploys the `ReceiverPays` contract, attaching enough Ether to cover the payments that will be made.
2. Alice authorises a payment by signing a message with their private key.
3. Alice sends the cryptographically signed message to Bob. The message does not need to be kept secret (explained later), and the mechanism for sending it does not matter.
4. Bob claims their payment by presenting the signed message to the smart contract, it verifies the authenticity of the message and then releases the funds.

Creating the signature

Alice does not need to interact with the Ethereum network to sign the transaction, the process is completely offline. In this tutorial, we will sign messages in the browser using web3.js and MetaMask, using the method described in EIP-762, as it provides a number of other security benefits.

```
/// Hashing first makes things easier
var hash = web3.utils.sha3("message to sign");
web3.eth.personal.sign(hash, web3.eth.defaultAccount, function () { console.log(
↪"Signed"); });
```

Note: The `web3.eth.personal.sign` prepends the length of the message to the signed data. Since we hash first, the message will always be exactly 32 bytes long, and thus this length prefix is always the same.

What to Sign

For a contract that fulfils payments, the signed message must include:

1. The recipient's address.
2. The amount to be transferred.
3. Protection against replay attacks.

A replay attack is when a signed message is reused to claim authorization for a second action. To avoid replay attacks we use the same as in Ethereum transactions themselves, a so-called nonce, which is the number of transactions sent by an account. The smart contract checks if a nonce is used multiple times.

Another type of replay attack can occur when the owner deploys a `ReceiverPays` smart contract, makes some payments, and then destroys the contract. Later, they decide to deploy the `RecipientPays` smart contract again, but the new contract does not know the nonces used in the previous deployment, so the attacker can use the old messages again.

Alice can protect against this attack by including the contract's address in the message, and only messages containing the contract's address itself will be accepted. You can find an example of this in the first two lines of the `claimPayment()` function of the full contract at the end of this section.

Packing arguments

Now that we have identified what information to include in the signed message, we are ready to put the message together, hash it, and sign it. For simplicity, we concatenate the data. The ethereumjs-abi library provides a function called `soliditySHA3` that mimics the behaviour of Solidity's `keccak256` function applied to arguments encoded using `abi.encodePacked`. Here is a JavaScript function that creates the proper signature for the `ReceiverPays` example:

```
// recipient is the address that should be paid.
// amount, in wei, specifies how much ether should be sent.
// nonce can be any unique number to prevent replay attacks
// contractAddress is used to prevent cross-contract replay attacks
function signPayment(recipient, amount, nonce, contractAddress, callback) {
    var hash = "0x" + abi.soliditySHA3(
        ["address", "uint256", "uint256", "address"],
        [recipient, amount, nonce, contractAddress]
    ).toString("hex");

    web3.eth.personal.sign(hash, web3.eth.defaultAccount, callback);
}
```

Recovering the Message Signer in Solidity

In general, ECDSA signatures consist of two parameters, r and s. Signatures in Ethereum include a third parameter called v, that you can use to verify which account's private key was used to sign the message, and the transaction's sender. Solidity provides a built-in function ecrecover that accepts a message along with the r, s and v parameters and returns the address that was used to sign the message.

Extracting the Signature Parameters

Signatures produced by web3.js are the concatenation of r, s and v, so the first step is to split these parameters apart. You can do this on the client-side, but doing it inside the smart contract means you only need to send one signature parameter rather than three. Splitting apart a byte array into its constituent parts is a mess, so we use inline assembly to do the job in the splitSignature function (the third function in the full contract at the end of this section).

Computing the Message Hash

The smart contract needs to know exactly what parameters were signed, and so it must recreate the message from the parameters and use that for signature verification. The functions prefixed and recoverSigner do this in the claimPayment function.

The full contract

```
pragma solidity >=0.4.24 <0.7.0;

contract ReceiverPays {
    address owner = msg.sender;

    mapping(uint256 => bool) usedNonces;

    constructor() public payable {}

    function claimPayment(uint256 amount, uint256 nonce, bytes memory signature)␣
↪public {
        require(!usedNonces[nonce]);
        usedNonces[nonce] = true;

        // this recreates the message that was signed on the client
        bytes32 message = prefixed(keccak256(abi.encodePacked(msg.sender, amount,␣
↪nonce, this)));

        require(recoverSigner(message, signature) == owner);

        msg.sender.transfer(amount);
    }

    /// destroy the contract and reclaim the leftover funds.
    function kill() public {
        require(msg.sender == owner);
        selfdestruct(msg.sender);
    }

    /// signature methods.
```

(continues on next page)

(continued from previous page)

```
    function splitSignature(bytes memory sig)
        internal
        pure
        returns (uint8 v, bytes32 r, bytes32 s)
    {
        require(sig.length == 65);

        assembly {
            // first 32 bytes, after the length prefix.
            r := mload(add(sig, 32))
            // second 32 bytes.
            s := mload(add(sig, 64))
            // final byte (first byte of the next 32 bytes).
            v := byte(0, mload(add(sig, 96)))
        }

        return (v, r, s);
    }

    function recoverSigner(bytes32 message, bytes memory sig)
        internal
        pure
        returns (address)
    {
        (uint8 v, bytes32 r, bytes32 s) = splitSignature(sig);

        return ecrecover(message, v, r, s);
    }

    /// builds a prefixed hash to mimic the behavior of eth_sign.
    function prefixed(bytes32 hash) internal pure returns (bytes32) {
        return keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32", hash));
    }
}
```

Writing a Simple Payment Channel

Alice now builds a simple but complete implementation of a payment channel. Payment channels use cryptographic signatures to make repeated transfers of Ether securely, instantaneously, and without transaction fees.

What is a Payment Channel?

Payment channels allow participants to make repeated transfers of Ether without using transactions. This means that you can avoid the delays and fees associated with transactions. We are going to explore a simple unidirectional payment channel between two parties (Alice and Bob). It involves three steps:

1. Alice funds a smart contract with Ether. This “opens” the payment channel.
2. Alice signs messages that specify how much of that Ether is owed to the recipient. This step is repeated for each payment.
3. Bob “closes” the payment channel, withdrawing their portion of the Ether and sending the remainder back to the sender.

Note: Only steps 1 and 3 require Ethereum transactions, step 2 means that the sender transmits a cryptographically signed message to the recipient via off chain methods (e.g. email). This means only two transactions are required to support any number of transfers.

Bob is guaranteed to receive their funds because the smart contract escrows the Ether and honours a valid signed message. The smart contract also enforces a timeout, so Alice is guaranteed to eventually recover their funds even if the recipient refuses to close the channel. It is up to the participants in a payment channel to decide how long to keep it open. For a short-lived transaction, such as paying an internet café for each minute of network access, the payment channel may be kept open for a limited duration. On the other hand, for a recurring payment, such as paying an employee an hourly wage, the payment channel may be kept open for several months or years.

Opening the Payment Channel

To open the payment channel, Alice deploys the smart contract, attaching the Ether to be escrowed and specifying the intended recipient and a maximum duration for the channel to exist. This is the function `SimplePaymentChannel` in the contract, at the end of this section.

Making Payments

Alice makes payments by sending signed messages to Bob. This step is performed entirely outside of the Ethereum network. Messages are cryptographically signed by the sender and then transmitted directly to the recipient.

Each message includes the following information:

- The smart contract's address, used to prevent cross-contract replay attacks.
- The total amount of Ether that is owed the recipient so far.

A payment channel is closed just once, at the end of a series of transfers. Because of this, only one of the messages sent is redeemed. This is why each message specifies a cumulative total amount of Ether owed, rather than the amount of the individual micropayment. The recipient will naturally choose to redeem the most recent message because that is the one with the highest total. The nonce per-message is not needed anymore, because the smart contract only honours a single message. The address of the smart contract is still used to prevent a message intended for one payment channel from being used for a different channel.

Here is the modified JavaScript code to cryptographically sign a message from the previous section:

```
function constructPaymentMessage(contractAddress, amount) {
    return abi.soliditySHA3(
        ["address", "uint256"],
        [contractAddress, amount]
    );
}

function signMessage(message, callback) {
    web3.eth.personal.sign(
        "0x" + message.toString("hex"),
        web3.eth.defaultAccount,
        callback
    );
}

// contractAddress is used to prevent cross-contract replay attacks.
// amount, in wei, specifies how much Ether should be sent.
```

(continues on next page)

(continued from previous page)

```
function signPayment(contractAddress, amount, callback) {
    var message = constructPaymentMessage(contractAddress, amount);
    signMessage(message, callback);
}
```

Closing the Payment Channel

When Bob is ready to receive their funds, it is time to close the payment channel by calling a `close` function on the smart contract. Closing the channel pays the recipient the Ether they are owed and destroys the contract, sending any remaining Ether back to Alice. To close the channel, Bob needs to provide a message signed by Alice.

The smart contract must verify that the message contains a valid signature from the sender. The process for doing this verification is the same as the process the recipient uses. The Solidity functions `isValidSignature` and `recoverSigner` work just like their JavaScript counterparts in the previous section, with the latter function borrowed from the `ReceiverPays` contract.

Only the payment channel recipient can call the `close` function, who naturally passes the most recent payment message because that message carries the highest total owed. If the sender were allowed to call this function, they could provide a message with a lower amount and cheat the recipient out of what they are owed.

The function verifies the signed message matches the given parameters. If everything checks out, the recipient is sent their portion of the Ether, and the sender is sent the rest via a `selfdestruct`. You can see the `close` function in the full contract.

Channel Expiration

Bob can close the payment channel at any time, but if they fail to do so, Alice needs a way to recover their escrowed funds. An *expiration* time was set at the time of contract deployment. Once that time is reached, Alice can call `claimTimeout` to recover their funds. You can see the `claimTimeout` function in the full contract.

After this function is called, Bob can no longer receive any Ether, so it is important that Bob closes the channel before the expiration is reached.

The full contract

```
pragma solidity >=0.4.24 <0.7.0;

contract SimplePaymentChannel {
    address payable public sender;      // The account sending payments.
    address payable public recipient;   // The account receiving the payments.
    uint256 public expiration;  // Timeout in case the recipient never closes.

    constructor (address payable _recipient, uint256 duration)
        public
        payable
    {
        sender = msg.sender;
        recipient = _recipient;
        expiration = now + duration;
    }
```

(continues on next page)

(continued from previous page)

```
    /// the recipient can close the channel at any time by presenting a
    /// signed amount from the sender. the recipient will be sent that amount,
    /// and the remainder will go back to the sender
    function close(uint256 amount, bytes memory signature) public {
        require(msg.sender == recipient);
        require(isValidSignature(amount, signature));

        recipient.transfer(amount);
        selfdestruct(sender);
    }

    /// the sender can extend the expiration at any time
    function extend(uint256 newExpiration) public {
        require(msg.sender == sender);
        require(newExpiration > expiration);

        expiration = newExpiration;
    }

    /// if the timeout is reached without the recipient closing the channel,
    /// then the Ether is released back to the sender.
    function claimTimeout() public {
        require(now >= expiration);
        selfdestruct(sender);
    }

    function isValidSignature(uint256 amount, bytes memory signature)
        internal
        view
        returns (bool)
    {
        bytes32 message = prefixed(keccak256(abi.encodePacked(this, amount)));

        // check that the signature is from the payment sender
        return recoverSigner(message, signature) == sender;
    }

    /// All functions below this are just taken from the chapter
    /// 'creating and verifying signatures' chapter.

    function splitSignature(bytes memory sig)
        internal
        pure
        returns (uint8 v, bytes32 r, bytes32 s)
    {
        require(sig.length == 65);

        assembly {
            // first 32 bytes, after the length prefix
            r := mload(add(sig, 32))
            // second 32 bytes
            s := mload(add(sig, 64))
            // final byte (first byte of the next 32 bytes)
            v := byte(0, mload(add(sig, 96)))
        }

        return (v, r, s);
```

(continues on next page)

(continued from previous page)

```
    }

    function recoverSigner(bytes32 message, bytes memory sig)
        internal
        pure
        returns (address)
    {
        (uint8 v, bytes32 r, bytes32 s) = splitSignature(sig);

        return ecrecover(message, v, r, s);
    }

    /// builds a prefixed hash to mimic the behavior of eth_sign.
    function prefixed(bytes32 hash) internal pure returns (bytes32) {
        return keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32", hash));
    }
}
```

Note: The function `splitSignature` does not use all security checks. A real implementation should use a more rigorously tested library, such as openzepplin's version of this code.

Verifying Payments

Unlike in the previous section, messages in a payment channel aren't redeemed right away. The recipient keeps track of the latest message and redeems it when it's time to close the payment channel. This means it's critical that the recipient perform their own verification of each message. Otherwise there is no guarantee that the recipient will be able to get paid in the end.

The recipient should verify each message using the following process:

1. Verify that the contact address in the message matches the payment channel.
2. Verify that the new total is the expected amount.
3. Verify that the new total does not exceed the amount of Ether escrowed.
4. Verify that the signature is valid and comes from the payment channel sender.

We'll use the ethereumjs-util library to write this verification. The final step can be done a number of ways, and we use JavaScript. The following code borrows the *constructMessage* function from the signing **JavaScript code** above:

```
// this mimics the prefixing behavior of the eth_sign JSON-RPC method.
function prefixed(hash) {
    return ethereumjs.ABI.soliditySHA3(
        ["string", "bytes32"],
        ["\x19Ethereum Signed Message:\n32", hash]
    );
}

function recoverSigner(message, signature) {
    var split = ethereumjs.Util.fromRpcSig(signature);
    var publicKey = ethereumjs.Util.ecrecover(message, split.v, split.r, split.s);
    var signer = ethereumjs.Util.pubToAddress(publicKey).toString("hex");
    return signer;
}
```

(continues on next page)

(continued from previous page)

```
function isValidSignature(contractAddress, amount, signature, expectedSigner) {
    var message = prefixed(constructPaymentMessage(contractAddress, amount));
    var signer = recoverSigner(message, signature);
    return signer.toLowerCase() ==
        ethereumjs.Util.stripHexPrefix(expectedSigner).toLowerCase();
}
```

2.3.5 Modular Contracts

A modular approach to building your contracts helps you reduce the complexity and improve the readability which will help to identify bugs and vulnerabilities during development and code review. If you specify and control the behaviour or each module in isolation, the interactions you have to consider are only those between the module specifications and not every other moving part of the contract. In the example below, the contract uses the `move` method of the `Balances` *library* to check that balances sent between addresses match what you expect. In this way, the `Balances` library provides an isolated component that properly tracks balances of accounts. It is easy to verify that the `Balances` library never produces negative balances or overflows and the sum of all balances is an invariant across the lifetime of the contract.

```
pragma solidity >=0.4.22 <0.7.0;

library Balances {
    function move(mapping(address => uint256) storage balances, address from, address␣
↪to, uint amount) internal {
        require(balances[from] >= amount);
        require(balances[to] + amount >= balances[to]);
        balances[from] -= amount;
        balances[to] += amount;
    }
}

contract Token {
    mapping(address => uint256) balances;
    using Balances for *;
    mapping(address => mapping (address => uint256)) allowed;

    event Transfer(address from, address to, uint amount);
    event Approval(address owner, address spender, uint amount);

    function transfer(address to, uint amount) public returns (bool success) {
        balances.move(msg.sender, to, amount);
        emit Transfer(msg.sender, to, amount);
        return true;

    }

    function transferFrom(address from, address to, uint amount) public returns (bool␣
↪success) {
        require(allowed[from][msg.sender] >= amount);
        allowed[from][msg.sender] -= amount;
        balances.move(from, to, amount);
        emit Transfer(from, to, amount);
        return true;
    }
```

(continues on next page)

(continued from previous page)

```
    function approve(address spender, uint tokens) public returns (bool success) {
        require(allowed[msg.sender][spender] == 0, "");
        allowed[msg.sender][spender] = tokens;
        emit Approval(msg.sender, spender, tokens);
        return true;
    }

    function balanceOf(address tokenOwner) public view returns (uint balance) {
        return balances[tokenOwner];
    }
}
```

2.4 Solidity in Depth

This section should provide you with all you need to know about Solidity.

2.4.1 Layout of a Solidity Source File

Source files can contain an arbitrary number of *contract definitions*, *import* directives, *pragma directives* and *struct* and *enum* definitions.

Pragmas

The `pragma` keyword is used to enable certain compiler features or checks. A pragma directive is always local to a source file, so you have to add the pragma to all your files if you want enable it in your whole project. If you *import* another file, the pragma from that file does *not* automatically apply to the importing file.

Version Pragma

Source files can (and should) be annotated with a version pragma to reject compilation with future compiler versions that might introduce incompatible changes. We try to keep these to an absolute minimum and introduce them in a way that changes in semantics also require changes in the syntax, but this is not always possible. Because of this, it is always a good idea to read through the changelog at least for releases that contain breaking changes. These releases always have versions of the form `0.x.0` or `x.0.0`.

The version pragma is used as follows: `pragma solidity ^0.5.2;`

A source file with the line above does not compile with a compiler earlier than version 0.5.2, and it also does not work on a compiler starting from version 0.6.0 (this second condition is added by using `^`). Because there will be no breaking changes until version `0.6.0`, you can be sure that your code compiles the way you intended. The exact version of the compiler is not fixed, so that bugfix releases are still possible.

It is possible to specify more complex rules for the compiler version, these follow the same syntax used by npm.

Note: Using the version pragma *does not* change the version of the compiler. It also *does not* enable or disable features of the compiler. It just instructs the compiler to check whether its version matches the one required by the pragma. If it does not match, the compiler issues an error.

Experimental Pragma

The second pragma is the experimental pragma. It can be used to enable features of the compiler or language that are not yet enabled by default. The following experimental pragmas are currently supported:

ABIEncoderV2

The new ABI encoder is able to encode and decode arbitrarily nested arrays and structs. It might produce less optimal code and has not received as much testing as the old encoder, but is considered non-experimental as of Solidity 0.6.0. You still have to explicitly activate it using `pragma experimental ABIEncoderV2;` - we kept the same pragma, even though it is not considered experimental anymore.

SMTChecker

This component has to be enabled when the Solidity compiler is built and therefore it is not available in all Solidity binaries. The *build instructions* explain how to activate this option. It is activated for the Ubuntu PPA releases in most versions, but not for the Docker images, Windows binaries or the statically-built Linux binaries. It can be activated for solc-js via the smtCallback if you have an SMT solver installed locally and run solc-js via node (not via the browser).

If you use `pragma experimental SMTChecker;`, then you get additional safety warnings which are obtained by querying an SMT solver. The component does not yet support all features of the Solidity language and likely outputs many warnings. In case it reports unsupported features, the analysis may not be fully sound.

Importing other Source Files

Syntax and Semantics

Solidity supports import statements to help modularise your code that are similar to those available in JavaScript (from ES6 on). However, Solidity does not support the concept of a default export.

At a global level, you can use import statements of the following form:

```
import "filename";
```

This statement imports all global symbols from “filename” (and symbols imported there) into the current global scope (different than in ES6 but backwards-compatible for Solidity). This form is not recommended for use, because it unpredictably pollutes the namespace. If you add new top-level items inside “filename”, they automatically appear in all files that import like this from “filename”. It is better to import specific symbols explicitly.

The following example creates a new global symbol `symbolName` whose members are all the global symbols from `"filename"`:

```
import * as symbolName from "filename";
```

which results in all global symbols being available in the format `symbolName.symbol`.

A variant of this syntax that is not part of ES6, but possibly useful is:

```
import "filename" as symbolName;
```

which is equivalent to `import * as symbolName from "filename";`.

If there is a naming collision, you can rename symbols while importing. For example, the code below creates new global symbols alias and symbol2 which reference symbol1 and symbol2 from inside "filename", respectively.

```
import {symbol1 as alias, symbol2} from "filename";
```

Paths

In the above, filename is always treated as a path with / as directory separator, and . as the current and .. as the parent directory. When . or .. is followed by a character except /, it is not considered as the current or the parent directory. All path names are treated as absolute paths unless they start with the current . or the parent directory ...

To import a file filename from the same directory as the current file, use import "./filename" as symbolName;. If you use import "filename" as symbolName; instead, a different file could be referenced (in a global "include directory").

It depends on the compiler (see *Use in Actual Compilers*) how to actually resolve the paths. In general, the directory hierarchy does not need to strictly map onto your local filesystem, and the path can also map to resources such as ipfs, http or git.

Note: Always use relative imports like import "./filename.sol"; and avoid using .. in path specifiers. In the latter case, it is probably better to use global paths and set up remappings as explained below.

Use in Actual Compilers

When invoking the compiler, you can specify how to discover the first element of a path, and also path prefix remappings. For example you can setup a remapping so that everything imported from the virtual directory github.com/ethereum/dapp-bin/library would actually be read from your local directory /usr/local/dapp-bin/library. If multiple remappings apply, the one with the longest key is tried first. An empty prefix is not allowed. The remappings can depend on a context, which allows you to configure packages to import e.g., different versions of a library of the same name.

solc:

For solc (the commandline compiler), you provide these path remappings as context:prefix=target arguments, where both the context: and the =target parts are optional (target defaults to prefix in this case). All remapping values that are regular files are compiled (including their dependencies).

This mechanism is backwards-compatible (as long as no filename contains = or :) and thus not a breaking change. All files in or below the context directory that import a file that starts with prefix are redirected by replacing prefix by target.

For example, if you clone github.com/ethereum/dapp-bin/ locally to /usr/local/dapp-bin, you can use the following in your source file:

```
import "github.com/ethereum/dapp-bin/library/iterable_mapping.sol" as it_mapping;
```

Then run the compiler:

```
solc github.com/ethereum/dapp-bin/=/usr/local/dapp-bin/ source.sol
```

As a more complex example, suppose you rely on a module that uses an old version of dapp-bin that you checked out to /usr/local/dapp-bin_old, then you can run:

```
solc module1:github.com/ethereum/dapp-bin/=/usr/local/dapp-bin/ \
     module2:github.com/ethereum/dapp-bin/=/usr/local/dapp-bin_old/ \
     source.sol
```

This means that all imports in `module2` point to the old version but imports in `module1` point to the new version.

Note: `solc` only allows you to include files from certain directories. They have to be in the directory (or subdirectory) of one of the explicitly specified source files or in the directory (or subdirectory) of a remapping target. If you want to allow direct absolute includes, add the remapping `/=/`.

If there are multiple remappings that lead to a valid file, the remapping with the longest common prefix is chosen.

Remix:

Remix provides an automatic remapping for GitHub and automatically retrieves the file over the network. You can import the iterable mapping as above, e.g.

```
import "github.com/ethereum/dapp-bin/library/iterable_mapping.sol" as it_mapping;
```

Remix may add other source code providers in the future.

Comments

Single-line comments (`//`) and multi-line comments (`/*...*/`) are possible.

```
// This is a single-line comment.

/*
This is a
multi-line comment.
*/
```

Note: A single-line comment is terminated by any unicode line terminator (LF, VF, FF, CR, NEL, LS or PS) in utf8 encoding. The terminator is still part of the source code after the comment, so if it is not an ascii symbol (these are NEL, LS and PS), it will lead to a parser error.

Additionally, there is another type of comment called a natspec comment, which is detailed in the *style guide*. They are written with a triple slash (`///`) or a double asterisk block(`/** ... */`) and they should be used directly above function declarations or statements. You can use Doxygen-style tags inside these comments to document functions, annotate conditions for formal verification, and provide a **confirmation text** which is shown to users when they attempt to invoke a function.

In the following example we document the title of the contract, the explanation for the two function parameters and two return variables.

```
pragma solidity >=0.4.0 <0.7.0;

/** @title Shape calculator. */
contract ShapeCalculator {
    /// @dev Calculates a rectangle's surface and perimeter.
    /// @param w Width of the rectangle.
    /// @param h Height of the rectangle.
    /// @return s The calculated surface.
```

(continues on next page)

(continued from previous page)

```
    /// @return p The calculated perimeter.
    function rectangle(uint w, uint h) public pure returns (uint s, uint p) {
        s = w * h;
        p = 2 * (w + h);
    }
}
```

2.4.2 Structure of a Contract

Contracts in Solidity are similar to classes in object-oriented languages. Each contract can contain declarations of *State Variables*, *Functions*, *Function Modifiers*, *Events*, *Struct Types* and *Enum Types*. Furthermore, contracts can inherit from other contracts.

There are also special kinds of contracts called *libraries* and *interfaces*.

The section about *contracts* contains more details than this section, which serves to provide a quick overview.

State Variables

State variables are variables whose values are permanently stored in contract storage.

```
pragma solidity >=0.4.0 <0.7.0;

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

See the *Types* section for valid state variable types and *Visibility and Getters* for possible choices for visibility.

Functions

Functions are the executable units of code within a contract.

```
pragma solidity >=0.4.0 <0.7.0;

contract SimpleAuction {
    function bid() public payable { // Function
        // ...
    }
}
```

Function Calls can happen internally or externally and have different levels of *visibility* towards other contracts. *Functions* accept *parameters and return variables* to pass parameters and values between them.

Function Modifiers

Function modifiers can be used to amend the semantics of functions in a declarative way (see *Function Modifiers* in the contracts section).

Overloading, that is, having the same modifier name with different parameters, is not possible.

Like functions, modifiers can be *overridden*.

```
pragma solidity >=0.4.22 <0.7.0;

contract Purchase {
    address public seller;

    modifier onlySeller() { // Modifier
        require(
            msg.sender == seller,
            "Only seller can call this."
        );
        _;
    }

    function abort() public view onlySeller { // Modifier usage
        // ...
    }
}
```

Events

Events are convenience interfaces with the EVM logging facilities.

```
pragma solidity >=0.4.21 <0.7.0;

contract SimpleAuction {
    event HighestBidIncreased(address bidder, uint amount); // Event

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

See *Events* in contracts section for information on how events are declared and can be used from within a dapp.

Struct Types

Structs are custom defined types that can group several variables (see *Structs* in types section).

```
pragma solidity >=0.4.0 <0.7.0;

contract Ballot {
    struct Voter { // Struct
        uint weight;
        bool voted;
        address delegate;
        uint vote;
    }
}
```

Enum Types

Enums can be used to create custom types with a finite set of ‘constant values’ (see *Enums* in types section).

```
pragma solidity >=0.4.0 <0.7.0;

contract Purchase {
    enum State { Created, Locked, Inactive } // Enum
}
```

2.4.3 Types

Solidity is a statically typed language, which means that the type of each variable (state and local) needs to be specified. Solidity provides several elementary types which can be combined to form complex types.

In addition, types can interact with each other in expressions containing operators. For a quick reference of the various operators, see *Order of Precedence of Operators*.

The concept of "undefined" or "null" values does not exist in Solidity, but newly declared variables always have a *default value* dependent on its type. To handle any unexpected values, you should use the *revert function* to revert the whole transaction, or return a tuple with a second *bool* value denoting success.

Value Types

The following types are also called value types because variables of these types will always be passed by value, i.e. they are always copied when they are used as function arguments or in assignments.

Booleans

`bool`: The possible values are constants `true` and `false`.

Operators:

- `!` (logical negation)
- `&&` (logical conjunction, "and")
- `||` (logical disjunction, "or")
- `==` (equality)
- `!=` (inequality)

The operators `||` and `&&` apply the common short-circuiting rules. This means that in the expression `f(x) || g(y)`, if `f(x)` evaluates to `true`, `g(y)` will not be evaluated even if it may have side-effects.

Integers

`int` / `uint`: Signed and unsigned integers of various sizes. Keywords `uint8` to `uint256` in steps of `8` (unsigned of 8 up to 256 bits) and `int8` to `int256`. `uint` and `int` are aliases for `uint256` and `int256`, respectively.

Operators:

- Comparisons: `<=`, `<`, `==`, `!=`, `>=`, `>` (evaluate to `bool`)
- Bit operators: `&`, `|`, `^` (bitwise exclusive or), `~` (bitwise negation)
- Shift operators: `<<` (left shift), `>>` (right shift)
- Arithmetic operators: `+`, `-`, unary `-`, `*`, `/`, `%` (modulo), `**` (exponentiation)

Warning: Integers in Solidity are restricted to a certain range. For example, with `uint32`, this is `0` up to `2**32 - 1`. If the result of some operation on those numbers does not fit inside this range, it is truncated. These truncations can have serious consequences that you should be aware of and mitigate against.

Comparisons

The value of a comparison is the one obtained by comparing the integer value.

Bit operations

Bit operations are performed on the two's complement representation of the number. This means that, for example `~int256(0) == int256(-1)`.

Shifts

The result of a shift operation has the type of the left operand, truncating the result to match the type.

- For positive and negative `x` values, `x << y` is equivalent to `x * 2**y`.
- For positive `x` values, `x >> y` is equivalent to `x / 2**y`.
- For negative `x` values, `x >> y` is equivalent to `(x + 1) / 2**y - 1` (which is the same as dividing `x` by `2**y` while rounding down towards negative infinity).
- In all cases, shifting by a negative `y` throws a runtime exception.

Warning: Before version `0.5.0` a right shift `x >> y` for negative `x` was equivalent to `x / 2**y`, i.e., right shifts used rounding up (towards zero) instead of rounding down (towards negative infinity).

Addition, Subtraction and Multiplication

Addition, subtraction and multiplication have the usual semantics. They wrap in two's complement representation, meaning that for example `uint256(0) - uint256(1) == 2**256 - 1`. You have to take these overflows into account when designing safe smart contracts.

The expression `-x` is equivalent to `(T(0) - x)` where `T` is the type of `x`. This means that `-x` will not be negative if the type of `x` is an unsigned integer type. Also, `-x` can be positive if `x` is negative. There is another caveat also resulting from two's complement representation:

```
int x = -2**255;
assert(-x == x);
```

This means that even if a number is negative, you cannot assume that its negation will be positive.

Division

Since the type of the result of an operation is always the type of one of the operands, division on integers always results in an integer. In Solidity, division rounds towards zero. This mean that `int256(-5) / int256(2) == int256(-2)`.

Note that in contrast, division on *literals* results in fractional values of arbitrary precision.

Note: Division by zero causes a failing assert.

Modulo

The modulo operation `a % n` yields the remainder `r` after the division of the operand `a` by the operand `n`, where `q = int(a / n)` and `r = a - (n * q)`. This means that modulo results in the same sign as its left operand (or zero) and `a % n == -(-a % n)` holds for negative `a`:

- `int256(5) % int256(2) == int256(1)`
- `int256(5) % int256(-2) == int256(1)`
- `int256(-5) % int256(2) == int256(-1)`
- `int256(-5) % int256(-2) == int256(-1)`

Note: Modulo with zero causes a failing assert.

Exponentiation

Exponentiation is only available for unsigned types in the exponent. The resulting type of an exponentiation is always equal to the type of the base. Please take care that it is large enough to hold the result and prepare for potential wrapping behaviour.

Note: Note that `0**0` is defined by the EVM as `1`.

Fixed Point Numbers

Warning: Fixed point numbers are not fully supported by Solidity yet. They can be declared, but cannot be assigned to or from.

`fixed` / `ufixed`: Signed and unsigned fixed point number of various sizes. Keywords `ufixedMxN` and `fixedMxN`, where `M` represents the number of bits taken by the type and `N` represents how many decimal points are available. `M` must be divisible by 8 and goes from 8 to 256 bits. `N` must be between 0 and 80, inclusive. `ufixed` and `fixed` are aliases for `ufixed128x18` and `fixed128x18`, respectively.

Operators:

- Comparisons: `<=`, `<`, `==`, `!=`, `>=`, `>` (evaluate to `bool`)
- Arithmetic operators: `+`, `-`, unary `-`, `*`, `/`, `%` (modulo)

Note: The main difference between floating point (`float` and `double` in many languages, more precisely IEEE 754 numbers) and fixed point numbers is that the number of bits used for the integer and the fractional part (the part after the decimal dot) is flexible in the former, while it is strictly defined in the latter. Generally, in floating point almost the entire space is used to represent the number, while only a small number of bits define where the decimal point is.

Address

The address type comes in two flavours, which are largely identical:

- `address`: Holds a 20 byte value (size of an Ethereum address).
- `address payable`: Same as `address`, but with the additional members `transfer` and `send`.

The idea behind this distinction is that `address payable` is an address you can send Ether to, while a plain `address` cannot be sent Ether.

Type conversions:

Implicit conversions from `address payable` to `address` are allowed, whereas conversions from `address` to `address payable` must be explicit via `payable(<address>)`.

Address literals can be implicitly converted to `address payable`.

Explicit conversions to and from `address` are allowed for integers, integer literals, `bytes20` and contract types with the following caveat: The result of a conversion of the form `address(x)` has the type `address payable`, if `x` is of integer or fixed bytes type, a literal or a contract with a receive or payable fallback function. If `x` is a contract without a receive or payable fallback function, then `address(x)` will be of type `address`. In external function signatures `address` is used for both the `address` and the `address payable` type.

Only expressions of type `address` can be converted to type `address payable` via `payable(<address>)`.

Note: It might very well be that you do not need to care about the distinction between `address` and `address payable` and just use `address` everywhere. For example, if you are using the *withdrawal pattern*, you can (and should) store the address itself as `address`, because you invoke the `transfer` function on `msg.sender`, which is an `address payable`.

Operators:

- `<=`, `<`, `==`, `!=`, `>=` and `>`

Warning: If you convert a type that uses a larger byte size to an `address`, for example `bytes32`, then the `address` is truncated. To reduce conversion ambiguity version 0.4.24 and higher of the compiler force you make the truncation explicit in the conversion. Take for example the 32-byte value `0x111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFFCCCC`.

You can use `address(uint160(bytes20(b)))`, which results in `0x111122223333444455556666777788889999aAaa`, or you can use `address(uint160(uint256(b)))`, which results in `0x777788889999AaAAbBbbCcccddDdeeeEfFFfCcCc`.

Note: The distinction between `address` and `address payable` was introduced with version 0.5.0. Also starting from that version, contracts do not derive from the address type, but can still be explicitly converted to `address` or to `address payable`, if they have a receive or payable fallback function.

Members of Addresses

For a quick reference of all members of address, see *Members of Address Types*.

- `balance` and `transfer`

It is possible to query the balance of an address using the property `balance` and to send Ether (in units of wei) to a payable address using the `transfer` function:

```
address payable x = address(0x123);
address myAddress = address(this);
if (x.balance < 10 && myAddress.balance >= 10) x.transfer(10);
```

The `transfer` function fails if the balance of the current contract is not large enough or if the Ether transfer is rejected by the receiving account. The `transfer` function reverts on failure.

Note: If `x` is a contract address, its code (more specifically: its *Receive Ether Function*, if present, or otherwise its *Fallback Function*, if present) will be executed together with the `transfer` call (this is a feature of the EVM and cannot be prevented). If that execution runs out of gas or fails in any way, the Ether transfer will be reverted and the current contract will stop with an exception.

- `send`

Send is the low-level counterpart of `transfer`. If the execution fails, the current contract will not stop with an exception, but `send` will return `false`.

Warning: There are some dangers in using `send`: The transfer fails if the call stack depth is at 1024 (this can always be forced by the caller) and it also fails if the recipient runs out of gas. So in order to make safe Ether transfers, always check the return value of `send`, use `transfer` or even better: use a pattern where the recipient withdraws the money.

- `call`, `delegatecall` and `staticcall`

In order to interface with contracts that do not adhere to the ABI, or to get more direct control over the encoding, the functions `call`, `delegatecall` and `staticcall` are provided. They all take a single `bytes memory` parameter and return the success condition (as a `bool`) and the returned data (`bytes memory`). The functions `abi.encode`, `abi.encodePacked`, `abi.encodeWithSelector` and `abi.encodeWithSignature` can be used to encode structured data.

Example:

```
bytes memory payload = abi.encodeWithSignature("register(string)", "MyName");
(bool success, bytes memory returnData) = address(nameReg).call(payload);
require(success);
```

Warning: All these functions are low-level functions and should be used with care. Specifically, any unknown contract might be malicious and if you call it, you hand over control to that contract which could in turn call back into your contract, so be prepared for changes to your state variables when the call returns. The regular way to interact with other contracts is to call a function on a contract object (`x.f()`).

Note: Previous versions of Solidity allowed these functions to receive arbitrary arguments and would also handle a first argument of type bytes4 differently. These edge cases were removed in version 0.5.0.

It is possible to adjust the supplied gas with the .gas() modifier:

```
address(nameReg).call.gas(1000000)(abi.encodeWithSignature("register(string)", "MyName
↪"));
```

Similarly, the supplied Ether value can be controlled too:

```
address(nameReg).call.value(1 ether)(abi.encodeWithSignature("register(string)",
↪"MyName"));
```

Lastly, these modifiers can be combined. Their order does not matter:

```
address(nameReg).call.gas(1000000).value(1 ether)(abi.encodeWithSignature(
↪"register(string)", "MyName"));
```

In a similar way, the function delegatecall can be used: the difference is that only the code of the given address is used, all other aspects (storage, balance, ...) are taken from the current contract. The purpose of delegatecall is to use library code which is stored in another contract. The user has to ensure that the layout of storage in both contracts is suitable for delegatecall to be used.

Note: Prior to homestead, only a limited variant called callcode was available that did not provide access to the original msg.sender and msg.value values. This function was removed in version 0.5.0.

Since byzantium staticcall can be used as well. This is basically the same as call, but will revert if the called function modifies the state in any way.

All three functions call, delegatecall and staticcall are very low-level functions and should only be used as a *last resort* as they break the type-safety of Solidity.

The .gas() option is available on all three methods, while the .value() option is not supported for delegatecall.

Note: All contracts can be converted to address type, so it is possible to query the balance of the current contract using address(this).balance.

Contract Types

Every *contract* defines its own type. You can implicitly convert contracts to contracts they inherit from. Contracts can be explicitly converted to and from the address type.

Explicit conversion to and from the address payable type is only possible if the contract type has a receive or payable fallback function. The conversion is still performed using address(x). If the contract type does not have a receive or payable fallback function, the conversion to address payable can be done using payable(address(x)). You can find more information in the section about the *address type*.

Note: Before version 0.5.0, contracts directly derived from the address type and there was no distinction between address and address payable.

If you declare a local variable of contract type (*MyContract c*), you can call functions on that contract. Take care to assign it from somewhere that is the same contract type.

You can also instantiate contracts (which means they are newly created). You can find more details in the *'Contracts via new'* section.

The data representation of a contract is identical to that of the `address` type and this type is also used in the *ABI*.

Contracts do not support any operators.

The members of contract types are the external functions of the contract including any state variables marked as `public`.

For a contract `C` you can use `type(C)` to access *type information* about the contract.

Fixed-size byte arrays

The value types `bytes1`, `bytes2`, `bytes3`, ..., `bytes32` hold a sequence of bytes from one to up to 32. `byte` is an alias for `bytes1`.

Operators:

- Comparisons: <=, <, ==, !=, >=, > (evaluate to `bool`)
- Bit operators: `&`, `|`, `^` (bitwise exclusive or), `~` (bitwise negation)
- Shift operators: << (left shift), >> (right shift)
- Index access: If `x` is of type `bytesI`, then `x[k]` for `0 <= k < I` returns the `k` th byte (read-only).

The shifting operator works with any integer type as right operand (but returns the type of the left operand), which denotes the number of bits to shift by. Shifting by a negative amount causes a runtime exception.

Members:

- `.length` yields the fixed length of the byte array (read-only).

Note: The type `byte[]` is an array of bytes, but due to padding rules, it wastes 31 bytes of space for each element (except in storage). It is better to use the `bytes` type instead.

Dynamically-sized byte array

`bytes`: Dynamically-sized byte array, see *Arrays*. Not a value-type!

`string`: Dynamically-sized UTF-8-encoded string, see *Arrays*. Not a value-type!

Address Literals

Hexadecimal literals that pass the address checksum test, for example `0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF` are of `address payable` type. Hexadecimal literals that are between 39 and 41 digits long and do not pass the checksum test produce an error. You can prepend (for integer types) or append (for bytesNN types) zeros to remove the error.

Note: The mixed-case address checksum format is defined in EIP-55.

Rational and Integer Literals

Integer literals are formed from a sequence of numbers in the range 0-9. They are interpreted as decimals. For example, `69` means sixty nine. Octal literals do not exist in Solidity and leading zeros are invalid.

Decimal fraction literals are formed by a `.` with at least one number on one side. Examples include `1.`, `.1` and `1.3`.

Scientific notation is also supported, where the base can have fractions and the exponent cannot. Examples include `2e10`, `-2e10`, `2e-10`, `2.5e1`.

Underscores can be used to separate the digits of a numeric literal to aid readability. For example, decimal `123_000`, hexadecimal `0x2eff_abde`, scientific decimal notation `1_2e345_678` are all valid. Underscores are only allowed between two digits and only one consecutive underscore is allowed. There is no additional semantic meaning added to a number literal containing underscores, the underscores are ignored.

Number literal expressions retain arbitrary precision until they are converted to a non-literal type (i.e. by using them together with a non-literal expression or by explicit conversion). This means that computations do not overflow and divisions do not truncate in number literal expressions.

For example, `(2**800 + 1) - 2**800` results in the constant `1` (of type `uint8`) although intermediate results would not even fit the machine word size. Furthermore, `.5 * 8` results in the integer `4` (although non-integers were used in between).

Any operator that can be applied to integers can also be applied to number literal expressions as long as the operands are integers. If any of the two is fractional, bit operations are disallowed and exponentiation is disallowed if the exponent is fractional (because that might result in a non-rational number).

Warning: Division on integer literals used to truncate in Solidity prior to version 0.4.0, but it now converts into a rational number, i.e. `5 / 2` is not equal to `2`, but to `2.5`.

Note: Solidity has a number literal type for each rational number. Integer literals and rational number literals belong to number literal types. Moreover, all number literal expressions (i.e. the expressions that contain only number literals and operators) belong to number literal types. So the number literal expressions `1 + 2` and `2 + 1` both belong to the same number literal type for the rational number three.

Note: Number literal expressions are converted into a non-literal type as soon as they are used with non-literal expressions. Disregarding types, the value of the expression assigned to `b` below evaluates to an integer. Because `a` is of type `uint128`, the expression `2.5 + a` has to have a proper type, though. Since there is no common type for the type of `2.5` and `uint128`, the Solidity compiler does not accept this code.

```
uint128 a = 1;
uint128 b = 2.5 + a + 0.5;
```

String Literals and Types

String literals are written with either double or single-quotes (`"foo"` or `'bar'`), and they can also be split into multiple consecutive parts (`"foo" "bar"` is equivalent to `"foobar"`) which can be helpful when dealing with long strings. They do not imply trailing zeroes as in C; `"foo"` represents three bytes, not four. As with integer literals, their type can vary, but they are implicitly convertible to `bytes1`, …, `bytes32`, if they fit, to `bytes` and to `string`.

For example, with `bytes32 samevar = "stringliteral"` the string literal is interpreted in its raw byte form when assigned to a `bytes32` type.

String literals support the following escape characters:

- `\<newline>` (escapes an actual newline)
- `\\` (backslash)
- `\'` (single quote)
- `\"` (double quote)
- `\b` (backspace)
- `\f` (form feed)
- `\n` (newline)
- `\r` (carriage return)
- `\t` (tab)
- `\v` (vertical tab)
- `\xNN` (hex escape, see below)
- `\uNNNN` (unicode escape, see below)

`\xNN` takes a hex value and inserts the appropriate byte, while `\uNNNN` takes a Unicode codepoint and inserts an UTF-8 sequence.

The string in the following example has a length of ten bytes. It starts with a newline byte, followed by a double quote, a single quote a backslash character and then (without separator) the character sequence `abcdef`.

```
"\n\"\'\\abc\
def"
```

Any unicode line terminator which is not a newline (i.e. LF, VF, FF, CR, NEL, LS, PS) is considered to terminate the string literal. Newline only terminates the string literal if it is not preceded by a `\`.

Hexadecimal Literals

Hexadecimal literals are prefixed with the keyword `hex` and are enclosed in double or single-quotes (`hex"001122FF"`, `hex'0011_22_FF'`). Their content must be hexadecimal digits which can optionally use a single underscore as separator between byte boundaries. The value of the literal will be the binary representation of the hexadecimal sequence.

Multiple hexadecimal literals separated by whitespace are concatenated into a single literal: `hex"00112233" hex"44556677"` is equivalent to `hex"0011223344556677"`

Hexadecimal literals behave like *string literals* and have the same convertibility restrictions.

Enums

Enums are one way to create a user-defined type in Solidity. They are explicitly convertible to and from all integer types but implicit conversion is not allowed. The explicit conversion from integer checks at runtime that the value lies inside the range of the enum and causes a failing assert otherwise. Enums require at least one member, and its default value when declared is the first member.

The data representation is the same as for enums in C: The options are represented by subsequent unsigned integer values starting from `0`.

```
pragma solidity >=0.4.16 <0.7.0;

contract test {
    enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill }
    ActionChoices choice;
    ActionChoices constant defaultChoice = ActionChoices.GoStraight;

    function setGoStraight() public {
        choice = ActionChoices.GoStraight;
    }

    // Since enum types are not part of the ABI, the signature of "getChoice"
    // will automatically be changed to "getChoice() returns (uint8)"
    // for all matters external to Solidity. The integer type used is just
    // large enough to hold all enum values, i.e. if you have more than 256 values,
    // `uint16` will be used and so on.
    function getChoice() public view returns (ActionChoices) {
        return choice;
    }

    function getDefaultChoice() public pure returns (uint) {
        return uint(defaultChoice);
    }
}
```

Note: Enums can also be declared on the file level, outside of contract or library definitions.

Function Types

Function types are the types of functions. Variables of function type can be assigned from functions and function parameters of function type can be used to pass functions to and return functions from function calls. Function types come in two flavours - *internal* and *external* functions:

Internal functions can only be called inside the current contract (more specifically, inside the current code unit, which also includes internal library functions and inherited functions) because they cannot be executed outside of the context of the current contract. Calling an internal function is realized by jumping to its entry label, just like when calling a function of the current contract internally.

External functions consist of an address and a function signature and they can be passed via and returned from external function calls.

Function types are notated as follows:

```
function (<parameter types>) {internal|external} [pure|view|payable] [returns (
↪<return types>)]
```

In contrast to the parameter types, the return types cannot be empty - if the function type should not return anything, the whole `returns (<return types>)` part has to be omitted.

By default, function types are internal, so the `internal` keyword can be omitted. Note that this only applies to function types. Visibility has to be specified explicitly for functions defined in contracts, they do not have a default.

Conversions:

A function type `A` is implicitly convertible to a function type `B` if and only if their parameter types are identical, their return types are identical, their internal/external property is identical and the state mutability of `A` is not more restrictive than the state mutability of `B`. In particular:

- `pure` functions can be converted to `view` and `non-payable` functions
- `view` functions can be converted to `non-payable` functions
- `payable` functions can be converted to `non-payable` functions

No other conversions between function types are possible.

The rule about `payable` and `non-payable` might be a little confusing, but in essence, if a function is `payable`, this means that it also accepts a payment of zero Ether, so it also is `non-payable`. On the other hand, a `non-payable` function will reject Ether sent to it, so `non-payable` functions cannot be converted to `payable` functions.

If a function type variable is not initialised, calling it results in a failed assertion. The same happens if you call a function after using `delete` on it.

If external function types are used outside of the context of Solidity, they are treated as the `function` type, which encodes the address followed by the function identifier together in a single `bytes24` type.

Note that public functions of the current contract can be used both as an internal and as an external function. To use `f` as an internal function, just use `f`, if you want to use its external form, use `this.f`.

Members:

External (or public) functions have the following members:

- `.address` returns the address of the contract of the function.
- `.selector` returns the *ABI function selector*
- `.gas(uint)` returns a callable function object which, when called, will send the specified amount of gas to the target function. See *External Function Calls* for more information.
- `.value(uint)` returns a callable function object which, when called, will send the specified amount of wei to the target function. See *External Function Calls* for more information.

Example that shows how to use the members:

```
pragma solidity >=0.4.16 <0.7.0;


contract Example {
    function f() public payable returns (bytes4) {
        assert(this.f.address == address(this));
        return this.f.selector;
    }

    function g() public {
        this.f.gas(10).value(800)();
    }
}
```

Example that shows how to use internal function types:

```
pragma solidity >=0.4.16 <0.7.0;


library ArrayUtils {
```

(continues on next page)

(continued from previous page)

```
    // internal functions can be used in internal library functions because
    // they will be part of the same code context
    function map(uint[] memory self, function (uint) pure returns (uint) f)
        internal
        pure
        returns (uint[] memory r)
    {
        r = new uint[](self.length);
        for (uint i = 0; i < self.length; i++) {
            r[i] = f(self[i]);
        }
    }

    function reduce(
        uint[] memory self,
        function (uint, uint) pure returns (uint) f
    )
        internal
        pure
        returns (uint r)
    {
        r = self[0];
        for (uint i = 1; i < self.length; i++) {
            r = f(r, self[i]);
        }
    }

    function range(uint length) internal pure returns (uint[] memory r) {
        r = new uint[](length);
        for (uint i = 0; i < r.length; i++) {
            r[i] = i;
        }
    }
}


contract Pyramid {
    using ArrayUtils for *;

    function pyramid(uint l) public pure returns (uint) {
        return ArrayUtils.range(l).map(square).reduce(sum);
    }

    function square(uint x) internal pure returns (uint) {
        return x * x;
    }

    function sum(uint x, uint y) internal pure returns (uint) {
        return x + y;
    }
}
```

Another example that uses external function types:

```
pragma solidity >=0.4.22 <0.7.0;

```

(continues on next page)

(continued from previous page)

```
contract Oracle {
    struct Request {
        bytes data;
        function(uint) external callback;
    }

    Request[] private requests;
    event NewRequest(uint);

    function query(bytes memory data, function(uint) external callback) public {
        requests.push(Request(data, callback));
        emit NewRequest(requests.length - 1);
    }

    function reply(uint requestID, uint response) public {
        // Here goes the check that the reply comes from a trusted source
        requests[requestID].callback(response);
    }
}


contract OracleUser {
    Oracle constant private ORACLE_CONST = Oracle(0x1234567); // known contract
    uint private exchangeRate;

    function buySomething() public {
        ORACLE_CONST.query("USD", this.oracleResponse);
    }

    function oracleResponse(uint response) public {
        require(
            msg.sender == address(ORACLE_CONST),
            "Only oracle can call this."
        );
        exchangeRate = response;
    }
}
```

Note: Lambda or inline functions are planned but not yet supported.

Reference Types

Values of reference type can be modified through multiple different names. Contrast this with value types where you get an independent copy whenever a variable of value type is used. Because of that, reference types have to be handled more carefully than value types. Currently, reference types comprise structs, arrays and mappings. If you use a reference type, you always have to explicitly provide the data area where the type is stored: `memory` (whose lifetime is limited to an external function call), `storage` (the location where the state variables are stored, where the lifetime is limited to the lifetime of a contract) or `calldata` (special data location that contains the function arguments, only available for external function call parameters).

An assignment or type conversion that changes the data location will always incur an automatic copy operation, while assignments inside the same data location only copy in some cases for storage types.

Data location

Every reference type has an additional annotation, the "data location", about where it is stored. There are three data locations: `memory`, `storage` and `calldata`. Calldata is only valid for parameters of external contract functions and is required for this type of parameter. Calldata is a non-modifiable, non-persistent area where function arguments are stored, and behaves mostly like memory.

Note: Prior to version 0.5.0 the data location could be omitted, and would default to different locations depending on the kind of variable, function type, etc., but all complex types must now give an explicit data location.

Data location and assignment behaviour

Data locations are not only relevant for persistency of data, but also for the semantics of assignments:

- Assignments between `storage` and `memory` (or from `calldata`) always create an independent copy.
- Assignments from `memory` to `memory` only create references. This means that changes to one memory variable are also visible in all other memory variables that refer to the same data.
- Assignments from `storage` to a **local** storage variable also only assign a reference.
- All other assignments to `storage` always copy. Examples for this case are assignments to state variables or to members of local variables of storage struct type, even if the local variable itself is just a reference.

```
pragma solidity >=0.4.0 <0.7.0;

contract C {
    // The data location of x is storage.
    // This is the only place where the
    // data location can be omitted.
    uint[] x;

    // The data location of memoryArray is memory.
    function f(uint[] memory memoryArray) public {
        x = memoryArray; // works, copies the whole array to storage
        uint[] storage y = x; // works, assigns a pointer, data location of y is storage
        y[7]; // fine, returns the 8th element
        y.pop(); // fine, modifies x through y
        delete x; // fine, clears the array, also modifies y
        // The following does not work; it would need to create a new temporary /
        // unnamed array in storage, but storage is "statically" allocated:
        // y = memoryArray;
        // This does not work either, since it would "reset" the pointer, but there
        // is no sensible location it could point to.
        // delete y;
        g(x); // calls g, handing over a reference to x
        h(x); // calls h and creates an independent, temporary copy in memory
    }

    function g(uint[] storage) internal pure {}
    function h(uint[] memory) public pure {}
}
```

Arrays

Arrays can have a compile-time fixed size, or they can have a dynamic size.

The type of an array of fixed size `k` and element type `T` is written as `T[k]`, and an array of dynamic size as `T[]`.

For example, an array of 5 dynamic arrays of `uint` is written as `uint[][5]`. The notation is reversed compared to some other languages. In Solidity, `X[3]` is always an array containing three elements of type `X`, even if `X` is itself an array. This is not the case in other languages such as C.

Indices are zero-based, and access is in the opposite direction of the declaration.

For example, if you have a variable `uint[][5] memory x`, you access the second `uint` in the third dynamic array using `x[2][1]`, and to access the third dynamic array, use `x[2]`. Again, if you have an array `T[5] a` for a type `T` that can also be an array, then `a[2]` always has type `T`.

Array elements can be of any type, including mapping or struct. The general restrictions for types apply, in that mappings can only be stored in the `storage` data location and publicly-visible functions need parameters that are *ABI types*.

It is possible to mark state variable arrays `public` and have Solidity create a *getter*. The numeric index becomes a required parameter for the getter.

Accessing an array past its end causes a failing assertion. Methods `.push()` and `.push(value)` can be used to append a new element at the end of the array, where `.push()` appends a zero-initialized element and returns a reference to it.

`bytes` and `strings` as Arrays

Variables of type `bytes` and `string` are special arrays. A `bytes` is similar to `byte[]`, but it is packed tightly in calldata and memory. `string` is equal to `bytes` but does not allow length or index access.

Solidity does not have string manipulation functions, but there are third-party string libraries. You can also compare two strings by their keccak256-hash using `keccak256(abi.encodePacked(s1)) == keccak256(abi.encodePacked(s2))` and concatenate two strings using `abi.encodePacked(s1, s2)`.

You should use `bytes` over `byte[]` because it is cheaper, since `byte[]` adds 31 padding bytes between the elements. As a general rule, use `bytes` for arbitrary-length raw byte data and `string` for arbitrary-length string (UTF-8) data. If you can limit the length to a certain number of bytes, always use one of the value types `bytes1` to `bytes32` because they are much cheaper.

Note: If you want to access the byte-representation of a string `s`, use `bytes(s).length` / `bytes(s)[7] = 'x';`. Keep in mind that you are accessing the low-level bytes of the UTF-8 representation, and not the individual characters.

Allocating Memory Arrays

Memory arrays with dynamic length can be created using the `new` operator. As opposed to storage arrays, it is **not** possible to resize memory arrays (e.g. the `.push` member functions are not available). You either have to calculate the required size in advance or create a new memory array and copy every element.

```
pragma solidity >=0.4.16 <0.7.0;

contract C {
```

(continues on next page)

(continued from previous page)

```
    function f(uint len) public pure {
        uint[] memory a = new uint[](7);
        bytes memory b = new bytes(len);
        assert(a.length == 7);
        assert(b.length == len);
        a[6] = 8;
    }
}
```

Array Literals

An array literal is a comma-separated list of one or more expressions, enclosed in square brackets (`[...]`). For example `[1, a, f(3)]`. There must be a common type all elements can be implicitly converted to. This is the elementary type of the array.

Array literals are always statically-sized memory arrays.

In the example below, the type of `[1, 2, 3]` is `uint8[3] memory`. Because the type of each of these constants is `uint8`, if you want the result to be a `uint[3] memory` type, you need to convert the first element to `uint`.

```
pragma solidity >=0.4.16 <0.7.0;

contract C {
    function f() public pure {
        g([uint(1), 2, 3]);
    }
    function g(uint[3] memory) public pure {
        // ...
    }
}
```

Fixed size memory arrays cannot be assigned to dynamically-sized memory arrays, i.e. the following is not possible:

```
pragma solidity >=0.4.0 <0.7.0;

// This will not compile.
contract C {
    function f() public {
        // The next line creates a type error because uint[3] memory
        // cannot be converted to uint[] memory.
        uint[] memory x = [uint(1), 3, 4];
    }
}
```

It is planned to remove this restriction in the future, but it creates some complications because of how arrays are passed in the ABI.

Array Members

length: Arrays have a `length` member that contains their number of elements. The length of memory arrays is fixed (but dynamic, i.e. it can depend on runtime parameters) once they are created.

push(): Dynamic storage arrays and `bytes` (not `string`) have a member function called `push()` that you can use to append a zero-initialised element at the end of the array. It returns a reference to the element, so that it can be used like `x.push().t = 2` or `x.push() = b`.

push(x): Dynamic storage arrays and `bytes` (not `string`) have a member function called `push(x)` that you can use to append a given element at the end of the array. The function returns nothing.

pop: Dynamic storage arrays and `bytes` (not `string`) have a member function called `pop` that you can use to remove an element from the end of the array. This also implicitly calls *delete* on the removed element.

Note: Increasing the length of a storage array by calling `push()` has constant gas costs because storage is zero-initialised, while decreasing the length by calling `pop()` has a cost that depends on the "size" of the element being removed. If that element is an array, it can be very costly, because it includes explicitly clearing the removed elements similar to calling *delete* on them.

Note: To use arrays of arrays in external (instead of public) functions, you need to activate ABIEncoderV2.

Note: In EVM versions before Byzantium, it was not possible to access dynamic arrays return from function calls. If you call functions that return dynamic arrays, make sure to use an EVM that is set to Byzantium mode.

```
pragma solidity >=0.4.16 <0.7.0;

contract ArrayContract {
    uint[2**20] m_aLotOfIntegers;
    // Note that the following is not a pair of dynamic arrays but a
    // dynamic array of pairs (i.e. of fixed size arrays of length two).
    // Because of that, T[] is always a dynamic array of T, even if T
    // itself is an array.
    // Data location for all state variables is storage.
    bool[2][] m_pairsOfFlags;

    // newPairs is stored in memory - the only possibility
    // for public contract function arguments
    function setAllFlagPairs(bool[2][] memory newPairs) public {
        // assignment to a storage array performs a copy of ``newPairs`` and
        // replaces the complete array ``m_pairsOfFlags``.
        m_pairsOfFlags = newPairs;
    }

    struct StructType {
        uint[] contents;
        uint moreInfo;
    }
    StructType s;

    function f(uint[] memory c) public {
        // stores a reference to ``s`` in ``g``
        StructType storage g = s;
        // also changes ``s.moreInfo``.
        g.moreInfo = 2;
        // assigns a copy because ``g.contents``
        // is not a local variable, but a member of
        // a local variable.
        g.contents = c;
    }
```

(continues on next page)

(continued from previous page)

```
    function setFlagPair(uint index, bool flagA, bool flagB) public {
        // access to a non-existing index will throw an exception
        m_pairsOfFlags[index][0] = flagA;
        m_pairsOfFlags[index][1] = flagB;
    }

    function changeFlagArraySize(uint newSize) public {
        // using push and pop is the only way to change the
        // length of an array
        if (newSize < m_pairsOfFlags.length) {
            while (m_pairsOfFlags.length > newSize)
                m_pairsOfFlags.pop();
        } else if (newSize > m_pairsOfFlags.length) {
            while (m_pairsOfFlags.length < newSize)
                m_pairsOfFlags.push();
        }
    }

    function clear() public {
        // these clear the arrays completely
        delete m_pairsOfFlags;
        delete m_aLotOfIntegers;
        // identical effect here
        m_pairsOfFlags = new bool[2][](0);
    }

    bytes m_byteData;

    function byteArrays(bytes memory data) public {
        // byte arrays ("bytes") are different as they are stored without padding,
        // but can be treated identical to "uint8[]"
        m_byteData = data;
        for (uint i = 0; i < 7; i++)
            m_byteData.push();
        m_byteData[3] = 0x08;
        delete m_byteData[2];
    }

    function addFlag(bool[2] memory flag) public returns (uint) {
        m_pairsOfFlags.push(flag);
        return m_pairsOfFlags.length;
    }

    function createMemoryArray(uint size) public pure returns (bytes memory) {
        // Dynamic memory arrays are created using `new`:
        uint[2][] memory arrayOfPairs = new uint[2][](size);

        // Inline arrays are always statically-sized and if you only
        // use literals, you have to provide at least one type.
        arrayOfPairs[0] = [uint(1), 2];

        // Create a dynamic byte array:
        bytes memory b = new bytes(200);
        for (uint i = 0; i < b.length; i++)
            b[i] = byte(uint8(i));
        return b;
    }
```

(continues on next page)

(continued from previous page)

```
}
```

Array Slices

Array slices are a view on a contiguous portion of an array. They are written as `x[start:end]`, where `start` and `end` are expressions resulting in a uint256 type (or implicitly convertible to it). The first element of the slice is `x[start]` and the last element is `x[end - 1]`.

If `start` is greater than `end` or if `end` is greater than the length of the array, an exception is thrown.

Both `start` and `end` are optional: `start` defaults to `0` and `end` defaults to the length of the array.

Array slices do not have any members. They are implicitly convertible to arrays of their underlying type and support index access. Index access is not absolute in the underlying array, but relative to the start of the slice.

Array slices do not have a type name which means no variable can have an array slices as type, they only exist in intermediate expressions.

Note: As of now, array slices are only implemented for calldata arrays.

Array slices are useful to ABI-decode secondary data passed in function parameters:

```
pragma solidity >=0.4.99 <0.7.0;

contract Proxy {
    /// Address of the client contract managed by proxy i.e., this contract
    address client;

    constructor(address _client) public {
        client = _client;
    }

    /// Forward call to "setOwner(address)" that is implemented by client
    /// after doing basic validation on the address argument.
    function forward(bytes calldata _payload) external {
        bytes4 sig = abi.decode(_payload[:4], (bytes4));
        if (sig == bytes4(keccak256("setOwner(address)"))) {
            address owner = abi.decode(_payload[4:], (address));
            require(owner != address(0), "Address of owner cannot be zero.");
        }
        (bool status,) = client.delegatecall(_payload);
        require(status, "Forwarded call failed.");
    }
}
```

Structs

Solidity provides a way to define new types in the form of structs, which is shown in the following example:

```
pragma solidity >=0.4.11 <0.7.0;

// Defines a new type with two fields.
// Declaring a struct outside of a contract allows
```

(continues on next page)

(continued from previous page)

```
// it to be shared by multiple contracts.
// Here, this is not really needed.
struct Funder {
    address addr;
    uint amount;
}

contract CrowdFunding {
    // Structs can also be defined inside contracts, which makes them
    // visible only there and in derived contracts.
    struct Campaign {
        address payable beneficiary;
        uint fundingGoal;
        uint numFunders;
        uint amount;
        mapping (uint => Funder) funders;
    }

    uint numCampaigns;
    mapping (uint => Campaign) campaigns;

    function newCampaign(address payable beneficiary, uint goal) public returns (uint␣
↪campaignID) {
        campaignID = numCampaigns++; // campaignID is return variable
        // Creates new struct in memory and copies it to storage.
        // We leave out the mapping type, because it is not valid in memory.
        // If structs are copied (even from storage to storage),
        // types that are not valid outside of storage (ex. mappings and array of␣
↪mappings)
        // are always omitted, because they cannot be enumerated.
        campaigns[campaignID] = Campaign(beneficiary, goal, 0, 0);
    }

    function contribute(uint campaignID) public payable {
        Campaign storage c = campaigns[campaignID];
        // Creates a new temporary memory struct, initialised with the given values
        // and copies it over to storage.
        // Note that you can also use Funder(msg.sender, msg.value) to initialise.
        c.funders[c.numFunders++] = Funder({addr: msg.sender, amount: msg.value});
        c.amount += msg.value;
    }

    function checkGoalReached(uint campaignID) public returns (bool reached) {
        Campaign storage c = campaigns[campaignID];
        if (c.amount < c.fundingGoal)
            return false;
        uint amount = c.amount;
        c.amount = 0;
        c.beneficiary.transfer(amount);
        return true;
    }
}
```

The contract does not provide the full functionality of a crowdfunding contract, but it contains the basic concepts necessary to understand structs. Struct types can be used inside mappings and arrays and they can itself contain mappings and arrays.

It is not possible for a struct to contain a member of its own type, although the struct itself can be the value type of a mapping member or it can contain a dynamically-sized array of its type. This restriction is necessary, as the size of the struct has to be finite.

Note how in all the functions, a struct type is assigned to a local variable with data location `storage`. This does not copy the struct but only stores a reference so that assignments to members of the local variable actually write to the state.

Of course, you can also directly access the members of the struct without assigning it to a local variable, as in `campaigns[campaignID].amount = 0`.

Mapping Types

Mapping types use the syntax `mapping(_KeyType => _ValueType)` and variables of mapping type are declared using the syntax `mapping(_KeyType => _ValueType) _VariableName`. The `_KeyType` can be any built-in value type plus `bytes` and `string`. User-defined or complex types such as contract types, enums, mappings, structs or array types apart from `bytes` and `string` are not allowed. `_ValueType` can be any type, including mappings, arrays and structs.

You can think of mappings as hash tables, which are virtually initialised such that every possible key exists and is mapped to a value whose byte-representation is all zeros, a type's *default value*. The similarity ends there, the key data is not stored in a mapping, only its `keccak256` hash is used to look up the value.

Because of this, mappings do not have a length or a concept of a key or value being set, and therefore cannot be erased without extra information regarding the assigned keys (see clearing-mappings).

Mappings can only have a data location of `storage` and thus are allowed for state variables, as storage reference types in functions, or as parameters for library functions. They cannot be used as parameters or return parameters of contract functions that are publicly visible.

You can mark state variables of mapping type as `public` and Solidity creates a *getter* for you. The `_KeyType` becomes a parameter for the getter. If `_ValueType` is a value type or a struct, the getter returns `_ValueType`. If `_ValueType` is an array or a mapping, the getter has one parameter for each `_KeyType`, recursively.

In the example below, the `MappingExample` contract defines a public `balances` mapping, with the key type an `address`, and a value type a `uint`, mapping an Ethereum address to an unsigned integer value. As `uint` is a value type, the getter returns a value that matches the type, which you can see in the `MappingUser` contract that returns the value at the specified address.

```
pragma solidity >=0.4.0 <0.7.0;

contract MappingExample {
    mapping(address => uint) public balances;

    function update(uint newBalance) public {
        balances[msg.sender] = newBalance;
    }
}

contract MappingUser {
    function f() public returns (uint) {
        MappingExample m = new MappingExample();
        m.update(100);
        return m.balances(address(this));
    }
}
```

The example below is a simplified version of an ERC20 token. _allowances is an example of a mapping type inside another mapping type. The example below uses _allowances to record the amount someone else is allowed to withdraw from your account.

```
pragma solidity >=0.4.0 <0.7.0;

contract MappingExample {

    mapping (address => uint256) private _balances;
    mapping (address => mapping (address => uint256)) private _allowances;

    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);

    function allowance(address owner, address spender) public view returns (uint256) {
        return _allowances[owner][spender];
    }

    function transferFrom(address sender, address recipient, uint256 amount) public␣
 ↪returns (bool) {
        _transfer(sender, recipient, amount);
        approve(sender, msg.sender, amount);
        return true;
    }

    function approve(address owner, address spender, uint256 amount) public returns␣
 ↪(bool) {
        require(owner != address(0), "ERC20: approve from the zero address");
        require(spender != address(0), "ERC20: approve to the zero address");

        _allowances[owner][spender] = amount;
        emit Approval(owner, spender, amount);
        return true;
    }

    function _transfer(address sender, address recipient, uint256 amount) internal {
        require(sender != address(0), "ERC20: transfer from the zero address");
        require(recipient != address(0), "ERC20: transfer to the zero address");

        _balances[sender] -= amount;
        _balances[recipient] += amount;
        emit Transfer(sender, recipient, amount);
    }
}
```

Iterable Mappings

You cannot iterate over mappings, i.e. you cannot enumerate their keys. It is possible, though, to implement a data structure on top of them and iterate over that. For example, the code below implements an IterableMapping library that the User contract then adds data too, and the sum function iterates over to sum all the values.

```
pragma solidity >=0.5.99 <0.7.0;

struct IndexValue { uint keyIndex; uint value; }
struct KeyFlag { uint key; bool deleted; }

```

(continues on next page)

(continued from previous page)

```
struct itmap {
    mapping(uint => IndexValue) data;
    KeyFlag[] keys;
    uint size;
}

library IterableMapping {
    function insert(itmap storage self, uint key, uint value) internal returns (bool␣
↪replaced) {
        uint keyIndex = self.data[key].keyIndex;
        self.data[key].value = value;
        if (keyIndex > 0)
            return true;
        else {
            self.keys.push();
            keyIndex = self.keys.length;
            self.data[key].keyIndex = keyIndex + 1;
            self.keys[keyIndex].key = key;
            self.size++;
            return false;
        }
    }

    function remove(itmap storage self, uint key) internal returns (bool success) {
        uint keyIndex = self.data[key].keyIndex;
        if (keyIndex == 0)
            return false;
        delete self.data[key];
        self.keys[keyIndex - 1].deleted = true;
        self.size --;
    }

    function contains(itmap storage self, uint key) internal view returns (bool) {
        return self.data[key].keyIndex > 0;
    }

    function iterate_start(itmap storage self) internal view returns (uint keyIndex) {
        return iterate_next(self, uint(-1));
    }

    function iterate_valid(itmap storage self, uint keyIndex) internal view returns␣
↪(bool) {
        return keyIndex < self.keys.length;
    }

    function iterate_next(itmap storage self, uint keyIndex) internal view returns␣
↪(uint r_keyIndex) {
        keyIndex++;
        while (keyIndex < self.keys.length && self.keys[keyIndex].deleted)
            keyIndex++;
        return keyIndex;
    }

    function iterate_get(itmap storage self, uint keyIndex) internal view returns␣
↪(uint key, uint value) {
        key = self.keys[keyIndex].key;
        value = self.data[key].value;
```

(continues on next page)

(continued from previous page)

```
    }
}

// How to use it
contract User {
    // Just a struct holding our data.
    itmap data;
    // Apply library functions to the data type.
    using IterableMapping for itmap;

    // Insert something
    function insert(uint k, uint v) public returns (uint size) {
        // This calls IterableMapping.insert(data, k, v)
        data.insert(k, v);
        // We can still access members of the struct,
        // but we should take care not to mess with them.
        return data.size;
    }

    // Computes the sum of all stored data.
    function sum() public view returns (uint s) {
        for (
            uint i = data.iterate_start();
            data.iterate_valid(i);
            i = data.iterate_next(i)
        ) {
            (, uint value) = data.iterate_get(i);
            s += value;
        }
    }
}
```

Operators Involving LValues

If `a` is an LValue (i.e. a variable or something that can be assigned to), the following operators are available as shorthands:

`a += e` is equivalent to `a = a + e`. The operators `-=`, `*=`, `/=`, `%=`, `|=`, `&=` and `^=` are defined accordingly. `a++` and `a--` are equivalent to `a += 1` / `a -= 1` but the expression itself still has the previous value of `a`. In contrast, `--a` and `++a` have the same effect on `a` but return the value after the change.

delete

`delete a` assigns the initial value for the type to `a`. I.e. for integers it is equivalent to `a = 0`, but it can also be used on arrays, where it assigns a dynamic array of length zero or a static array of the same length with all elements set to their initial value. `delete a[x]` deletes the item at index `x` of the array and leaves all other elements and the length of the array untouched. This especially means that it leaves a gap in the array. If you plan to remove items, a *mapping* is probably a better choice.

For structs, it assigns a struct with all members reset. In other words, the value of `a` after `delete a` is the same as if `a` would be declared without assignment, with the following caveat:

`delete` has no effect on mappings (as the keys of mappings may be arbitrary and are generally unknown). So if you delete a struct, it will reset all members that are not mappings and also recurse into the members unless they are mappings. However, individual keys and what they map to can be deleted: If a is a mapping, then delete a[x] will delete the value stored at x.

It is important to note that delete a really behaves like an assignment to a, i.e. it stores a new object in a. This distinction is visible when a is reference variable: It will only reset a itself, not the value it referred to previously.

```
pragma solidity >=0.4.0 <0.7.0;

contract DeleteExample {
    uint data;
    uint[] dataArray;

    function f() public {
        uint x = data;
        delete x; // sets x to 0, does not affect data
        delete data; // sets data to 0, does not affect x
        uint[] storage y = dataArray;
        delete dataArray; // this sets dataArray.length to zero, but as uint[] is a␣
↪complex object, also
        // y is affected which is an alias to the storage object
        // On the other hand: "delete y" is not valid, as assignments to local␣
↪variables
        // referencing storage objects can only be made from existing storage␣
↪objects.
        assert(y.length == 0);
    }
}
```

Conversions between Elementary Types

Implicit Conversions

An implicit type conversion is automatically applied by the compiler in some cases during assignments, when passing arguments to functions and when applying operators. In general, an implicit conversion between value-types is possible if it makes sense semantically and no information is lost.

For example, uint8 is convertible to uint16 and int128 to int256, but int8 is not convertible to uint256, because uint256 cannot hold values such as -1.

If an operator is applied to different types, the compiler tries to implicitly convert one of the operands to the type of the other (the same is true for assignments). This means that operations are always performed in the type of one of the operands.

For more details about which implicit conversions are possible, please consult the sections about the types themselves.

In the example below, y and z, the operands of the addition, do not have the same type, but uint8 can be implicitly converted to uint16 and not vice-versa. Because of that, y is converted to the type of z before the addition is performed in the uint16 type. The resulting type of the expression y + z is uint16`. Because it is assigned to a variable of type ``uint32 another implicit conversion is performed after the addition.

```
uint8 y;
uint16 z;
uint32 x = y + z;
```

Explicit Conversions

If the compiler does not allow implicit conversion but you are confident a conversion will work, an explicit type conversion is sometimes possible. This may result in unexpected behaviour and allows you to bypass some security features of the compiler, so be sure to test that the result is what you want and expect!

Take the following example that converts a negative `int` to a `uint`:

```
int  y = -3;
uint x = uint(y);
```

At the end of this code snippet, `x` will have the value `0xfffff..fd` (64 hex characters), which is -3 in the two's complement representation of 256 bits.

If an integer is explicitly converted to a smaller type, higher-order bits are cut off:

```
uint32 a = 0x12345678;
uint16 b = uint16(a); // b will be 0x5678 now
```

If an integer is explicitly converted to a larger type, it is padded on the left (i.e., at the higher order end). The result of the conversion will compare equal to the original integer:

```
uint16 a = 0x1234;
uint32 b = uint32(a); // b will be 0x00001234 now
assert(a == b);
```

Fixed-size bytes types behave differently during conversions. They can be thought of as sequences of individual bytes and converting to a smaller type will cut off the sequence:

```
bytes2 a = 0x1234;
bytes1 b = bytes1(a); // b will be 0x12
```

If a fixed-size bytes type is explicitly converted to a larger type, it is padded on the right. Accessing the byte at a fixed index will result in the same value before and after the conversion (if the index is still in range):

```
bytes2 a = 0x1234;
bytes4 b = bytes4(a); // b will be 0x12340000
assert(a[0] == b[0]);
assert(a[1] == b[1]);
```

Since integers and fixed-size byte arrays behave differently when truncating or padding, explicit conversions between integers and fixed-size byte arrays are only allowed, if both have the same size. If you want to convert between integers and fixed-size byte arrays of different size, you have to use intermediate conversions that make the desired truncation and padding rules explicit:

```
bytes2 a = 0x1234;
uint32 b = uint16(a); // b will be 0x00001234
uint32 c = uint32(bytes4(a)); // c will be 0x12340000
uint8 d = uint8(uint16(a)); // d will be 0x34
uint8 e = uint8(bytes1(a)); // e will be 0x12
```

Conversions between Literals and Elementary Types

Integer Types

Decimal and hexadecimal number literals can be implicitly converted to any integer type that is large enough to represent it without truncation:

```
uint8 a = 12; // fine
uint32 b = 1234; // fine
uint16 c = 0x123456; // fails, since it would have to truncate to 0x3456
```

Fixed-Size Byte Arrays

Decimal number literals cannot be implicitly converted to fixed-size byte arrays. Hexadecimal number literals can be, but only if the number of hex digits exactly fits the size of the bytes type. As an exception both decimal and hexadecimal literals which have a value of zero can be converted to any fixed-size bytes type:

```
bytes2 a = 54321; // not allowed
bytes2 b = 0x12; // not allowed
bytes2 c = 0x123; // not allowed
bytes2 d = 0x1234; // fine
bytes2 e = 0x0012; // fine
bytes4 f = 0; // fine
bytes4 g = 0x0; // fine
```

String literals and hex string literals can be implicitly converted to fixed-size byte arrays, if their number of characters matches the size of the bytes type:

```
bytes2 a = hex"1234"; // fine
bytes2 b = "xy"; // fine
bytes2 c = hex"12"; // not allowed
bytes2 d = hex"123"; // not allowed
bytes2 e = "x"; // not allowed
bytes2 f = "xyz"; // not allowed
```

Addresses

As described in *Address Literals*, hex literals of the correct size that pass the checksum test are of `address` type. No other literals can be implicitly converted to the `address` type.

Explicit conversions from `bytes20` or any integer type to `address` result in `address payable`.

An `address a` can be converted to `address payable` via `payable(a)`.

2.4.4 Units and Globally Available Variables

PlatON Units

A literal number can take a suffix of `von`, `finney`, `szabo` or `lat` to specify a subdenomination of PlatON, where PlatON numbers without a postfix are assumed to be von.

```
* ``assert(1 von == 1);``
* ``assert(1 szabo == 1e12);``
* ``assert(1 finney == 1e15);``
* ``assert(1 lat == 1e18);``
```

The only effect of the subdenomination suffix is a multiplication by a power of ten.

Time Units

Suffixes like `seconds`, `minutes`, `hours`, `days` and `weeks` after literal numbers can be used to specify units of time where seconds are the base unit and units are considered naively in the following way:

- `1 == 1 seconds`
- `1 minutes == 60 seconds`
- `1 hours == 60 minutes`
- `1 days == 24 hours`
- `1 weeks == 7 days`

Take care if you perform calendar calculations using these units, because not every year equals 365 days and not even every day has 24 hours because of leap seconds. Due to the fact that leap seconds cannot be predicted, an exact calendar library has to be updated by an external oracle.

Note: The suffix `years` has been removed in version 0.5.0 due to the reasons above.

These suffixes cannot be applied to variables. For example, if you want to interpret a function parameter in days, you can in the following way:

```
function f(uint start, uint daysAfter) public {
    if (now >= start + daysAfter * 1 days) {
      // ...
    }
}
```

Special Variables and Functions

There are special variables and functions which always exist in the global namespace and are mainly used to provide information about the blockchain or are general-use utility functions.

Block and Transaction Properties

- `blockhash(uint blockNumber) returns (bytes32)`: hash of the given block - only works for 256 most recent, excluding current, blocks
- `block.coinbase` (`address payable`): current block miner's address
- `block.difficulty` (`uint`): current block difficulty
- `block.gaslimit` (`uint`): current block gaslimit
- `block.number` (`uint`): current block number
- `block.timestamp` (`uint`): current block timestamp as seconds since unix epoch
- `gasleft() returns (uint256)`: remaining gas
- `msg.data` (`bytes calldata`): complete calldata
- `msg.sender` (`address payable`): sender of the message (current call)

- `msg.sig` (`bytes4`): first four bytes of the calldata (i.e. function identifier)
- `msg.value` (`uint`): number of wei sent with the message
- `now` (`uint`): current block timestamp (alias for `block.timestamp`)
- `tx.gasprice` (`uint`): gas price of the transaction
- `tx.origin` (`address payable`): sender of the transaction (full call chain)

Note: The values of all members of `msg`, including `msg.sender` and `msg.value` can change for every **external** function call. This includes calls to library functions.

Note: Do not rely on `block.timestamp`, `now` and `blockhash` as a source of randomness, unless you know what you are doing.

Both the timestamp and the block hash can be influenced by miners to some degree. Bad actors in the mining community can for example run a casino payout function on a chosen hash and just retry a different hash if they did not receive any money.

The current block timestamp must be strictly larger than the timestamp of the last block, but the only guarantee is that it will be somewhere between the timestamps of two consecutive blocks in the canonical chain.

Note: The block hashes are not available for all blocks for scalability reasons. You can only access the hashes of the most recent 256 blocks, all other values will be zero.

Note: The function `blockhash` was previously known as `block.blockhash`, which was deprecated in version 0.4.22 and removed in version 0.5.0.

Note: The function `gasleft` was previously known as `msg.gas`, which was deprecated in version 0.4.21 and removed in version 0.5.0.

ABI Encoding and Decoding Functions

- `abi.decode(bytes memory encodedData, (...)) returns (...)`: ABI-decodes the given data, while the types are given in parentheses as second argument. Example: `(uint a, uint[2] memory b, bytes memory c) = abi.decode(data, (uint, uint[2], bytes))`
- `abi.encode(...) returns (bytes memory)`: ABI-encodes the given arguments
- `abi.encodePacked(...) returns (bytes memory)`: Performs *packed encoding* of the given arguments. Note that packed encoding can be ambiguous!
- `abi.encodeWithSelector(bytes4 selector, ...) returns (bytes memory)`: ABI-encodes the given arguments starting from the second and prepends the given four-byte selector
- `abi.encodeWithSignature(string memory signature, ...) returns (bytes memory)`: Equivalent to `abi.encodeWithSelector(bytes4(keccak256(bytes(signature))), ...)``

Note: These encoding functions can be used to craft data for external function calls without actually calling an external function. Furthermore, `keccak256(abi.encodePacked(a, b))` is a way to compute the hash of structured data (although be aware that it is possible to craft a "hash collision" using different function parameter types).

See the documentation about the *ABI* and the *tightly packed encoding* for details about the encoding.

Error Handling

See the dedicated section on *assert and require* for more details on error handling and when to use which function.

`assert(bool condition)`: causes an invalid opcode and thus state change reversion if the condition is not met - to be used for internal errors.

`require(bool condition)`: reverts if the condition is not met - to be used for errors in inputs or external components.

`require(bool condition, string memory message)`: reverts if the condition is not met - to be used for errors in inputs or external components. Also provides an error message.

`revert()`: abort execution and revert state changes

`revert(string memory reason)`: abort execution and revert state changes, providing an explanatory string

Mathematical and Cryptographic Functions

`addmod(uint x, uint y, uint k) returns (uint)`: compute `(x + y) % k` where the addition is performed with arbitrary precision and does not wrap around at `2**256`. Assert that `k != 0` starting from version 0.5.0.

`mulmod(uint x, uint y, uint k) returns (uint)`: compute `(x * y) % k` where the multiplication is performed with arbitrary precision and does not wrap around at `2**256`. Assert that `k != 0` starting from version 0.5.0.

`keccak256(bytes memory) returns (bytes32)`: compute the Keccak-256 hash of the input

Note: There used to be an alias for `keccak256` called `sha3`, which was removed in version 0.5.0.

`sha256(bytes memory) returns (bytes32)`: compute the SHA-256 hash of the input

`ripemd160(bytes memory) returns (bytes20)`: compute RIPEMD-160 hash of the input

`ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)`:
recover the address associated with the public key from elliptic curve signature or return zero on error. The function parameters correspond to ECDSA values of the signature:

`r` = first 32 bytes of signature `s` = second 32 bytes of signature `v` = final 1 byte of signature

`ecrecover` returns an `address`, and not an `address payable`. See *address payable* for conversion, in case you need to transfer funds to the recovered address.

For further details, read example usage.

Warning: If you use `ecrecover`, be aware that a valid signature can be turned into a different valid signature without requiring knowledge of the corresponding private key. In the Homestead hard fork, this issue was fixed for _transaction_ signatures (see EIP-2), but the ecrecover function remained unchanged.

This is usually not a problem unless you require signatures to be unique or use them to identify items. OpenZeppelin have a ECDSA helper library that you can use as a wrapper for `ecrecover` without this issue.

Note: When running `sha256`, `ripemd160` or `ecrecover` on a *private blockchain*, you might encounter Out-of-Gas. This is because these functions are implemented as "precompiled contracts" and only really exist after they receive the first message (although their contract code is hardcoded). Messages to non-existing contracts are more expensive and thus the execution might run into an Out-of-Gas error. A workaround for this problem is to first send Wei (1 for example) to each of the contracts before you use them in your actual contracts. This is not an issue on the main or test net.

Members of Address Types

`<address>.balance (uint256)`: balance of the *Address* in Wei

`<address payable>.transfer(uint256 amount)`: send given amount of Wei to *Address*, reverts on failure, forwards 2300 gas stipend, not adjustable

`<address payable>.send(uint256 amount) returns (bool)`: send given amount of Wei to *Address*, returns `false` on failure, forwards 2300 gas stipend, not adjustable

`<address>.call(bytes memory) returns (bool, bytes memory)`: issue low-level `CALL` with the given payload, returns success condition and return data, forwards all available gas, adjustable

`<address>.delegatecall(bytes memory) returns (bool, bytes memory)`: issue low-level `DELEGATECALL` with the given payload, returns success condition and return data, forwards all available gas, adjustable

`<address>.staticcall(bytes memory) returns (bool, bytes memory)`: issue low-level `STATICCALL` with the given payload, returns success condition and return data, forwards all available gas, adjustable

For more information, see the section on *Address*.

Warning: You should avoid using `.call()` whenever possible when executing another contract function as it bypasses type checking, function existence check, and argument packing.

Warning: There are some dangers in using `send`: The transfer fails if the call stack depth is at 1024 (this can always be forced by the caller) and it also fails if the recipient runs out of gas. So in order to make safe Ether transfers, always check the return value of `send`, use `transfer` or even better: Use a pattern where the recipient withdraws the money.

Note: Prior to version 0.5.0, Solidity allowed address members to be accessed by a contract instance, for example `this.balance`. This is now forbidden and an explicit conversion to address must be done: `address(this).balance`.

Note: If state variables are accessed via a low-level delegatecall, the storage layout of the two contracts must align in order for the called contract to correctly access the storage variables of the calling contract by name. This is of course not the case if storage pointers are passed as function arguments as in the case for the high-level libraries.

Note: Prior to version 0.5.0, `.call`, `.delegatecall` and `.staticcall` only returned the success condition and not the return data.

Note: Prior to version 0.5.0, there was a member called `callcode` with similar but slightly different semantics than `delegatecall`.

Contract Related

`this` (current contract's type): the current contract, explicitly convertible to *Address*

`selfdestruct(address payable recipient)`: Destroy the current contract, sending its funds to the given *Address* and end execution. Note that `selfdestruct` has some peculiarities inherited from the EVM:

- the receiving contract's receive function is not executed.
- the contract is only really destroyed at the end of the transaction and `revert` s might "undo" the destruction.

Furthermore, all functions of the current contract are callable directly including the current function.

Note: Prior to version 0.5.0, there was a function called `suicide` with the same semantics as `selfdestruct`.

Type Information

The expression `type(X)` can be used to retrieve information about the type `X`. Currently, there is limited support for this feature, but it might be expanded in the future. The following properties are available for a contract type `C`:

`type(C).name`: The name of the contract.

`type(C).creationCode`: Memory byte array that contains the creation bytecode of the contract. This can be used in inline assembly to build custom creation routines, especially by using the `create2` opcode. This property can **not** be accessed in the contract itself or any derived contract. It causes the bytecode to be included in the bytecode of the call site and thus circular references like that are not possible.

`type(C).runtimeCode`: Memory byte array that contains the runtime bytecode of the contract. This is the code that is usually deployed by the constructor of `C`. If `C` has a constructor that uses inline assembly, this might be different from the actually deployed bytecode. Also note that libraries modify their runtime bytecode at time of deployment to guard against regular calls. The same restrictions as with `.creationCode` also apply for this property.

2.4.5 Expressions and Control Structures

Control Structures

Most of the control structures known from curly-braces languages are available in Solidity:

There is: `if`, `else`, `while`, `do`, `for`, `break`, `continue`, `return`, with the usual semantics known from C or JavaScript.

Solidity also supports exception handling in the form of `try`/`catch`-statements, but only for *external function calls* and contract creation calls.

Parentheses can *not* be omitted for conditionals, but curly brances can be omitted around single-statement bodies.

Note that there is no type conversion from non-boolean to boolean types as there is in C and JavaScript, so `if (1) { ... }` is *not* valid Solidity.

Function Calls

Internal Function Calls

Functions of the current contract can be called directly ("internally"), also recursively, as seen in this nonsensical example:

```
pragma solidity >=0.4.16 <0.7.0;

contract C {
    function g(uint a) public pure returns (uint ret) { return a + f(); }
    function f() internal pure returns (uint ret) { return g(7) + f(); }
}
```

These function calls are translated into simple jumps inside the EVM. This has the effect that the current memory is not cleared, i.e. passing memory references to internally-called functions is very efficient. Only functions of the same contract instance can be called internally.

You should still avoid excessive recursion, as every internal function call uses up at least one stack slot and there are only 1024 slots available.

External Function Calls

The expressions `this.g(8);` and `c.g(2);` (where `c` is a contract instance) are also valid function calls, but this time, the function will be called "externally", via a message call and not directly via jumps. Please note that function calls on `this` cannot be used in the constructor, as the actual contract has not been created yet.

Functions of other contracts have to be called externally. For an external call, all function arguments have to be copied to memory.

Note: A function call from one contract to another does not create its own transaction, it is a message call as part of the overall transaction.

When calling functions of other contracts, you can specify the amount of Wei or gas sent with the call with the special options `.value()` and `.gas()`, respectively. Any Wei you send to the contract is added to the total balance of the contract:

```
pragma solidity >=0.4.0 <0.7.0;

contract InfoFeed {
    function info() public payable returns (uint ret) { return 42; }
}

contract Consumer {
    InfoFeed feed;
    function setFeed(InfoFeed addr) public { feed = addr; }
    function callFeed() public { feed.info.value(10).gas(800)(); }
}
```

You need to use the modifier `payable` with the `info` function because otherwise, the `.value()` option would not be available.

Warning: Be careful that `feed.info.value(10).gas(800)` only locally sets the `value` and amount of `gas` sent with the function call, and the parentheses at the end perform the actual call. So in this case, the function is not called and the `value` and `gas` settings are lost.

Function calls cause exceptions if the called contract does not exist (in the sense that the account does not contain code) or if the called contract itself throws an exception or goes out of gas.

Warning: Any interaction with another contract imposes a potential danger, especially if the source code of the contract is not known in advance. The current contract hands over control to the called contract and that may potentially do just about anything. Even if the called contract inherits from a known parent contract, the inheriting contract is only required to have a correct interface. The implementation of the contract, however, can be completely arbitrary and thus, pose a danger. In addition, be prepared in case it calls into other contracts of your system or even back into the calling contract before the first call returns. This means that the called contract can change state variables of the calling contract via its functions. Write your functions in a way that, for example, calls to external functions happen after any changes to state variables in your contract so your contract is not vulnerable to a reentrancy exploit.

Named Calls and Anonymous Function Parameters

Function call arguments can be given by name, in any order, if they are enclosed in `{ }` as can be seen in the following example. The argument list has to coincide by name with the list of parameters from the function declaration, but can be in arbitrary order.

```
pragma solidity >=0.4.0 <0.7.0;

contract C {
    mapping(uint => uint) data;

    function f() public {
        set({value: 2, key: 3});
    }

    function set(uint key, uint value) public {
        data[key] = value;
    }
```

(continues on next page)

(continued from previous page)

```
}
```

Omitted Function Parameter Names

The names of unused parameters (especially return parameters) can be omitted. Those parameters will still be present on the stack, but they are inaccessible.

```
pragma solidity >=0.4.16 <0.7.0;

contract C {
    // omitted name for parameter
    function func(uint k, uint) public pure returns(uint) {
        return k;
    }
}
```

Creating Contracts via `new`

A contract can create other contracts using the `new` keyword. The full code of the contract being created has to be known when the creating contract is compiled so recursive creation-dependencies are not possible.

```
pragma solidity >=0.5.0 <0.7.0;

contract D {
    uint public x;
    constructor(uint a) public payable {
        x = a;
    }
}

contract C {
    D d = new D(4); // will be executed as part of C's constructor

    function createD(uint arg) public {
        D newD = new D(arg);
        newD.x();
    }

    function createAndEndowD(uint arg, uint amount) public payable {
        // Send ether along with the creation
        D newD = (new D).value(amount)(arg);
        newD.x();
    }
}
```

As seen in the example, it is possible to send Ether while creating an instance of `D` using the `.value()` option, but it is not possible to limit the amount of gas. If the creation fails (due to out-of-stack, not enough balance or other problems), an exception is thrown.

Order of Evaluation of Expressions

The evaluation order of expressions is not specified (more formally, the order in which the children of one node in the expression tree are evaluated is not specified, but they are of course evaluated before the node itself). It is only guaranteed that statements are executed in order and short-circuiting for boolean expressions is done.

Assignment

Destructuring Assignments and Returning Multiple Values

Solidity internally allows tuple types, i.e. a list of objects of potentially different types whose number is a constant at compile-time. Those tuples can be used to return multiple values at the same time. These can then either be assigned to newly declared variables or to pre-existing variables (or LValues in general).

Tuples are not proper types in Solidity, they can only be used to form syntactic groupings of expressions.

```
pragma solidity >0.4.23 <0.7.0;

contract C {
    uint index;

    function f() public pure returns (uint, bool, uint) {
        return (7, true, 2);
    }

    function g() public {
        // Variables declared with type and assigned from the returned tuple,
        // not all elements have to be specified (but the number must match).
        (uint x, , uint y) = f();
        // Common trick to swap values -- does not work for non-value storage types.
        (x, y) = (y, x);
        // Components can be left out (also for variable declarations).
        (index, , ) = f(); // Sets the index to 7
    }
}
```

It is not possible to mix variable declarations and non-declaration assignments, i.e. the following is not valid: `(x, uint y) = (1, 2);`

Note: Prior to version 0.5.0 it was possible to assign to tuples of smaller size, either filling up on the left or on the right side (which ever was empty). This is now disallowed, so both sides have to have the same number of components.

Warning: Be careful when assigning to multiple variables at the same time when reference types are involved, because it could lead to unexpected copying behaviour.

Complications for Arrays and Structs

The semantics of assignments are a bit more complicated for non-value types like arrays and structs. Assigning *to* a state variable always creates an independent copy. On the other hand, assigning to a local variable creates an independent copy only for elementary types, i.e. static types that fit into 32 bytes. If structs or arrays (including `bytes` and `string`) are assigned from a state variable to a local variable, the local variable holds a reference to the original state variable. A second assignment to the local variable does not modify the state but only changes the reference. Assignments to members (or elements) of the local variable *do* change the state.

In the example below the call to g(x) has no effect on x because it creates an independent copy of the storage value in memory. However, h(x) successfully modifies x because only a reference and not a copy is passed.

```
pragma solidity >=0.4.16 <0.7.0;

contract C {
    uint[20] x;

    function f() public {
        g(x);
        h(x);
    }

    function g(uint[20] memory y) internal pure {
        y[2] = 3;
    }

    function h(uint[20] storage y) internal {
        y[3] = 4;
    }
}
```

Scoping and Declarations

A variable which is declared will have an initial default value whose byte-representation is all zeros. The "default values" of variables are the typical "zero-state" of whatever the type is. For example, the default value for a bool is false. The default value for the uint or int types is 0. For statically-sized arrays and bytes1 to bytes32, each individual element will be initialized to the default value corresponding to its type. For dynamically-sized arrays, bytes and string, the default value is an empty array or string. For the enum type, the default value is its first member.

Scoping in Solidity follows the widespread scoping rules of C99 (and many other languages): Variables are visible from the point right after their declaration until the end of the smallest { }-block that contains the declaration. As an exception to this rule, variables declared in the initialization part of a for-loop are only visible until the end of the for-loop.

Variables that are parameter-like (function parameters, modifier parameters, catch parameters, ...) are visible inside the code block that follows - the body of the function/modifier for a function and modifier parameter and the catch block for a catch parameter.

Variables and other items declared outside of a code block, for example functions, contracts, user-defined types, etc., are visible even before they were declared. This means you can use state variables before they are declared and call functions recursively.

As a consequence, the following examples will compile without warnings, since the two variables have the same name but disjoint scopes.

```
pragma solidity >=0.5.0 <0.7.0;
contract C {
    function minimalScoping() pure public {
        {
            uint same;
            same = 1;
        }
```

(continues on next page)

(continued from previous page)

```
        {
            uint same;
            same = 3;
        }
    }
}
```

As a special example of the C99 scoping rules, note that in the following, the first assignment to `x` will actually assign the outer and not the inner variable. In any case, you will get a warning about the outer variable being shadowed.

```
pragma solidity >=0.5.0 <0.7.0;
// This will report a warning
contract C {
    function f() pure public returns (uint) {
        uint x = 1;
        {
            x = 2; // this will assign to the outer variable
            uint x;
        }
        return x; // x has value 2
    }
}
```

> **Warning:** Before version 0.5.0 Solidity followed the same scoping rules as JavaScript, that is, a variable declared anywhere within a function would be in scope for the entire function, regardless where it was declared. The following example shows a code snippet that used to compile but leads to an error starting from version 0.5.0.

```
pragma solidity >=0.5.0 <0.7.0;
// This will not compile
contract C {
    function f() pure public returns (uint) {
        x = 2;
        uint x;
        return x;
    }
}
```

Error handling: Assert, Require, Revert and Exceptions

Solidity uses state-reverting exceptions to handle errors. Such an exception undoes all changes made to the state in the current call (and all its sub-calls) and flags an error to the caller.

When exceptions happen in a sub-call, they "bubble up" (i.e., exceptions are rethrown) automatically. Exceptions to this rule are `send` and the low-level functions `call`, `delegatecall` and `staticcall`: they return `false` as their first return value in case of an exception instead of "bubbling up".

> **Warning:** The low-level functions `call`, `delegatecall` and `staticcall` return `true` as their first return value if the account called is non-existent, as part of the design of the EVM. Account existence must be checked prior to calling if needed.

Exceptions can be caught with the `try/catch` statement.

`assert` and `require`

The convenience functions `assert` and `require` can be used to check for conditions and throw an exception if the condition is not met.

The `assert` function should only be used to test for internal errors, and to check invariants. Properly functioning code should never reach a failing `assert` statement; if this happens there is a bug in your contract which you should fix. Language analysis tools can evaluate your contract to identify the conditions and function calls which will reach a failing `assert`.

An `assert`-style exception is generated in the following situations:

1. If you access an array or an array slice at a too large or negative index (i.e. `x[i]` where `i >= x.length` or `i < 0`).
2. If you access a fixed-length `bytesN` at a too large or negative index.
3. If you divide or modulo by zero (e.g. `5 / 0` or `23 % 0`).
4. If you shift by a negative amount.
5. If you convert a value too big or negative into an enum type.
6. If you call a zero-initialized variable of internal function type.
7. If you call `assert` with an argument that evaluates to false.

The `require` function should be used to ensure valid conditions that cannot be detected until execution time. This includes conditions on inputs or return values from calls to external contracts.

A `require`-style exception is generated in the following situations:

1. Calling `require` with an argument that evaluates to `false`.
2. If you call a function via a message call but it does not finish properly (i.e., it runs out of gas, has no matching function, or throws an exception itself), except when a low level operation `call`, `send`, `delegatecall`, `callcode` or `staticcall` is used. The low level operations never throw exceptions but indicate failures by returning `false`.
3. If you create a contract using the `new` keyword but the contract creation *does not finish properly*.
4. If you perform an external function call targeting a contract that contains no code.
5. If your contract receives Ether via a public function without `payable` modifier (including the constructor and the fallback function).
6. If your contract receives Ether via a public getter function.
7. If a `.transfer()` fails.

You can optionally provide a message string for `require`, but not for `assert`.

The following example shows how you can use `require` to check conditions on inputs and `assert` for internal error checking.

```
pragma solidity >=0.5.0 <0.7.0;

contract Sharer {
    function sendHalf(address payable addr) public payable returns (uint balance) {
        require(msg.value % 2 == 0, "Even value required.");
        uint balanceBeforeTransfer = address(this).balance;
```

(continues on next page)

(continued from previous page)

```
        addr.transfer(msg.value / 2);
        // Since transfer throws an exception on failure and
        // cannot call back here, there should be no way for us to
        // still have half of the money.
        assert(address(this).balance == balanceBeforeTransfer - msg.value / 2);
        return address(this).balance;
    }
}
```

Internally, Solidity performs a revert operation (instruction `0xfd`) for a `require`-style exception and executes an invalid operation (instruction `0xfe`) to throw an `assert`-style exception. In both cases, this causes the EVM to revert all changes made to the state. The reason for reverting is that there is no safe way to continue execution, because an expected effect did not occur. Because we want to keep the atomicity of transactions, the safest action is to revert all changes and make the whole transaction (or at least call) without effect.

In both cases, the caller can react on such failures using `try`/`catch` (in the failing `assert`-style exception only if enough gas is left), but the changes in the caller will always be reverted.

Note: `assert`-style exceptions consume all gas available to the call, while `require`-style exceptions do not consume any gas starting from the Metropolis release.

`revert`

The `revert` function is another way to trigger exceptions from within other code blocks to flag an error and revert the current call. The function takes an optional string message containing details about the error that is passed back to the caller.

The following example shows how to use an error string together with `revert` and the equivalent `require`:

```
pragma solidity >=0.5.0 <0.7.0;

contract VendingMachine {
    function buy(uint amount) public payable {
        if (amount > msg.value / 2 ether)
            revert("Not enough Ether provided.");
        // Alternative way to do it:
        require(
            amount <= msg.value / 2 ether,
            "Not enough Ether provided."
        );
        // Perform the purchase.
    }
}
```

The two syntax options are equivalent, it's developer preference which to use.

The provided string is *abi-encoded* as if it were a call to a function `Error(string)`. In the above example, `revert("Not enough Ether provided.");` returns the following hexadecimal as error return data:

```
0x08c379a0                                                         // Function␣
↪selector for Error(string)
0x0000000000000000000000000000000000000000000000000000000000000020 // Data offset
0x000000000000000000000000000000000000000000000000000000000000001a // String length
0x4e6f7420656e6f7567682045746865722070726f76696465642e000000000000 // String data
```

The provided message can be retrieved by the caller using `try`/`catch` as shown below.

Note: There used to be a keyword called `throw` with the same semantics as `revert()` which was deprecated in version 0.4.13 and removed in version 0.5.0.

`try`/`catch`

A failure in an external call can be caught using a try/catch statement, as follows:

```
pragma solidity ^0.6.0;

interface DataFeed { function getData(address token) external returns (uint value); }

contract FeedConsumer {
    DataFeed feed;
    uint errorCount;
    function rate(address token) public returns (uint value, bool success) {
        // Permanently disable the mechanism if there are
        // more than 10 errors.
        require(errorCount < 10);
        try feed.getData(token) returns (uint v) {
            return (v, true);
        } catch Error(string memory /*reason*/) {
            // This is executed in case
            // revert was called inside getData
            // and a reason string was provided.
            errorCount++;
            return (0, false);
        } catch (bytes memory /*lowLevelData*/) {
            // This is executed in case revert() was used
            // or there was a failing assertion, division
            // by zero, etc. inside getData.
            errorCount++;
            return (0, false);
        }
    }
}
```

The `try` keyword has to be followed by an expression representing an external function call or a contract creation (`new ContractName()`). Errors inside the expression are not caught (for example if it is a complex expression that also involves internal function calls), only a revert happening inside the external call itself. The `returns` part (which is optional) that follows declares return variables matching the types returned by the external call. In case there was no error, these variables are assigned and the contract's execution continues inside the first success block. If the end of the success block is reached, execution continues after the `catch` blocks.

Currently, Solidity supports different kinds of catch blocks depending on the type of error. If the error was caused by `revert("reasonString")` or `require(false, "reasonString")` (or an internal error that causes such an exception), then the catch clause of the type `catch Error(string memory reason)` will be executed.

It is planned to support other types of error data in the future. The string `Error` is currently parsed as is and is not treated as an identifier.

The clause `catch (bytes memory lowLevelData)` is executed if the error signature does not match any other clause, there was an error during decoding of the error message, if there was a failing assertion in the external call (for example due to a division by zero or a failing `assert()`) or if no error data was provided with the exception. The declared variable provides access to the low-level error data in that case.

If you are not interested in the error data, you can just use `catch { ... }` (even as the only catch clause).

In order to catch all error cases, you have to have at least the clause `catch { ...}` or the clause `catch (bytes memory lowLevelData) { ... }`.

The variables declared in the `returns` and the `catch` clause are only in scope in the block that follows.

Note: If an error happens during the decoding of the return data inside a try/catch-statement, this causes an exception in the currently executing contract and because of that, it is not caught in the catch clause. If there is an error during decoding of `catch Error(string memory reason)` and there is a low-level catch clause, this error is caught there.

Note: If execution reaches a catch-block, then the state-changing effects of the external call have been reverted. If execution reaches the success block, the effects were not reverted. If the effects have been reverted, then execution either continues in a catch block or the execution of the try/catch statement itself reverts (for example due to decoding failures as noted above or due to not providing a low-level catch clause).

2.4.6 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.

```
pragma solidity >=0.4.22 <0.7.0;


contract OwnedToken {
    // `TokenCreator` is a contract type that is defined below.
    // It is fine to reference it as long as it is not used
```

(continues on next page)

(continued from previous page)

```
    // 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) public {
        // 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.
```

(continues on next page)

(continued from previous page)

```
        // 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). External functions are sometimes more efficient when they receive large arrays of data, because the data is not copied from calldata to memory.

`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`.

`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.

```
pragma solidity >=0.4.16 <0.7.0;

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`.

```
pragma solidity >=0.4.0 <0.7.0;

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.

```
pragma solidity >=0.4.0 <0.7.0;

contract C {
    uint public data = 42;
}

contract Caller {
    C c = new C();
```

(continues on next page)

(continued from previous page)

```
    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.

```
pragma solidity >=0.4.0 <0.7.0;

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 `data(0)`. If you want to return an entire array in one call, then you need to write a function, for example:

```
pragma solidity >=0.4.0 <0.7.0;

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:

```
pragma solidity >=0.4.0 <0.7.0;

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

It generates a function of the following form. The mapping in the struct is omitted because there is no good way to provide the key for the mapping:

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

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*.

```
pragma solidity >=0.5.0 <0.7.0;

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

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

(continues on next page)

(continued from previous page)

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

    constructor(uint initialPrice) public { 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;
    }
}
```

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

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

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.

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 State Variables

State variables can be declared as `constant`. In this case, they have to be assigned from an expression which is a constant at compile time. Any expression that accesses storage, blockchain data (e.g. `now`, `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.

The compiler does not reserve a storage slot for these variables, and every occurrence is replaced by the respective constant expression (which might be computed to a single value by the optimizer).

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

```
pragma solidity >=0.4.0 <0.7.0;

contract C {
    uint constant x = 32**22 + 8;
    string constant text = "abc";
    bytes32 constant myHash = keccak256("abc");
}
```

Functions

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:

```
pragma solidity >=0.4.16 <0.7.0;

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 new `ABIEncoderV2` feature by adding `pragma experimental ABIEncoderV2;` 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:

```
pragma solidity >=0.4.16 <0.7.0;

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 using `return;`, or you can provide return values (either a single or *multiple ones*) directly with the `return` statement:

```
pragma solidity >=0.4.16 <0.7.0;

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

This form is equivalent to first assigning values to the return variables and then using `return;` to leave the function.

Note: You cannot return some types from non-internal functions, notably multi-dimensional dynamic arrays and structs. If you enable the new `ABIEncoderV2` feature by adding `pragma experimental ABIEncoderV2;` 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*.

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.

```
pragma solidity >=0.5.0 <0.7.0;

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

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.

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.

```
pragma solidity >=0.5.0 <0.7.0;

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.

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 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 fallback 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`.

```
pragma solidity ^0.6.0;

// 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 `fallback () external [payable]` (without the `function` keyword). This function cannot have arguments, cannot return anything and must have `external` visibility. It 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`.

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: Even though the fallback function cannot have arguments, one can still use `msg.data` to retrieve any payload supplied with the call. After having checked the first four bytes of `msg.data`, you can use `abi.decode` together with the array slice syntax to decode ABI-encoded data: `(c, d) = abi.decode(msg.data[4:], (uint256, uint256));` Note that this should only be used as a last resort and proper functions should be used instead.

```
pragma solidity ^0.6.0;

contract Test {
    // 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; }
    uint x;
}

contract TestPayable {
    // 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; }
    uint x;
    uint y;
}

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);
```

(continues on next page)

(continued from previous page)

```
    }

    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.
        require(address(test).send(2 ether));
        // results in test.x becoming == 2 and test.y becoming 2 ether.
    }
}
```

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`.

```
pragma solidity >=0.4.16 <0.7.0;

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.

```
pragma solidity >=0.4.16 <0.7.0;

// 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;
    }
}
```

(continues on next page)

(continued from previous page)

```
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.

```
pragma solidity >=0.4.16 <0.7.0;

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. If you use arrays (including `string` and `bytes`) as indexed arguments, its Keccak-256 hash is stored as a topic instead, this is because a topic can only hold a single word (32 bytes).

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.

```
pragma solidity >=0.4.21 <0.7.0;

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 event = clientReceipt.Deposit();

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

(continues on next page)

(continued from previous page)

```
// Or pass a callback to start watching immediately
var event = 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"]
   }
}
```

Low-Level Interface to Logs

It is also possible to access the low-level interface to the logging mechanism via the functions `log0`, `log1`, `log2`, `log3` and `log4`. Each function `logi` takes `i + 1` parameter of type `bytes32`, where the first argument will be used for the data part of the log and the others as topics. The event call above can be performed in the same way as

```
pragma solidity >=0.4.10 <0.7.0;

contract C {
    function f() public payable {
        uint256 _id = 0x420042;
        log3(
            bytes32(msg.value),
            ␣
↪bytes32(0x50cb9fe53daa9737b786ab3646f04d0150dc50ef4e75f59509d83667ad5adb20),
            bytes32(uint256(msg.sender)),
            bytes32(_id)
        );
    }
}
```

where the long hexadecimal number is equal to `keccak256("Deposit(address,bytes32,uint256)")`, the signature of the event.

Additional Resources for Understanding Events

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

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.

```
pragma solidity ^0.6.0;


contract Owned {
    constructor() public { owner = 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 Mortal is Owned {
    // The keyword `virtual` means that the function can change
    // its behaviour in derived classes ("overriding").
    function kill() 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;
}
```

(continues on next page)

(continued from previous page)

```


// Multiple inheritance is possible. Note that `owned` is
// also a base class of `mortal`, yet there is only a single
// instance of `owned` (as for virtual inheritance in C++).
contract Named is Owned, Mortal {
    constructor(bytes32 name) public {
        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 kill() 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.
            Mortal.kill();
        }
    }
}


// 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, Mortal, 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 `kill` anymore.
    function kill() public override(Mortal, Named) { Named.kill(); }
    function get() public view returns(uint r) { return info; }

    uint info;
}
```

Note that above, we call `mortal.kill()` to “forward” the destruction request. The way this is done is problematic, as seen in the following example:

```
pragma solidity ^0.6.0;

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

(continues on next page)

(continued from previous page)

```
}

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

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

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

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

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

```
pragma solidity >=0.4.22 <0.7.0;

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

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

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


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

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

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.kill()` (note that the final inheritance sequence is – starting with the most derived contract: Final, Base2, Base1, mortal, 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 as shown in this example:

```
pragma solidity >=0.5.0 <0.7.0;

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

contract Middle is Base {}

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

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:

```
pragma solidity >=0.5.0 <0.7.0;

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.

```
pragma solidity >=0.5.0 <0.7.0;

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`.

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

```
pragma solidity >=0.5.0 <0.7.0;

contract A
{
    function f() external pure 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:

```
pragma solidity >=0.5.0 <0.7.0;

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:

```
pragma solidity >=0.5.0 <0.7.0;

contract Base1
```

(continues on next page)

(continued from previous page)

```
{
    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 zero 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.

Constructor functions can be either `public` or `internal`. If there is no constructor, the contract will assume the default constructor, which is equivalent to `constructor() public {}`. For example:

```
pragma solidity >=0.5.0 <0.7.0;

contract A {
    uint public a;

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

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

A constructor set as `internal` causes the contract to be marked as *abstract*.

> **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.

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:

```
pragma solidity >=0.4.22 <0.7.0;

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

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

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

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".

```
pragma solidity >=0.4.0 <0.7.0;

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:

```
pragma solidity >=0.4.0 <0.7.0;

contract Base1 {
    constructor() public {}
}

contract Base2 {
    constructor() public {}
}

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

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

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

```
pragma solidity >=0.4.0 <0.7.0;

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:

```
pragma solidity ^0.6.0;

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

contract Cat is Feline {
    function utterance() public 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".

Interfaces

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

- They cannot inherit other contracts or interfaces.
- All declared functions must be external.
- They cannot declare a constructor.
- They cannot declare state variables.

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:

```
pragma solidity >=0.5.0 <0.7.0;

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`, which means that they can be overridden. This does not automatically mean that an overriding function can be overridden again - this is only possible if the overriding function is marked `virtual`.

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 (`L.f()` if `L` is the name of the library). Furthermore, `internal` functions of libraries are visible in all contracts, just as if the library were a base contract. 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`.

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).

```
pragma solidity >=0.4.22 <0.7.0;


// 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
```

(continues on next page)

(continued from previous page)

```
    // 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:

```
pragma solidity >=0.4.16 <0.7.0;

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);
            r.limbs[i] = a + b + carry;
            if (a + b < a || (a + b == uint(-1) && 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(uint(-1));
        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 cannot know where the library will be deployed at, these addresses have to be filled into the final bytecode by a linker (see *Using the Commandline Compiler* for how to use the commandline compiler for linking). If the addresses are not given as arguments to the compiler, the compiled hex code will contain placeholders of the form `__Set______` (where `Set` is the name of the library). The address can be filled manually by replacing all those 40 symbols by the hex encoding of the address of the library contract.

Note: Manually linking libraries on the generated bytecode is discouraged, because in this way, the library name is restricted to 36 characters. You should ask the compiler to link the libraries at the time a contract is compiled by either using the `--libraries` option of `solc` or the `libraries` key if you use the standard-JSON interface to the compiler.

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 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:

```
pragma solidity >0.5.13 <0.7.0;

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

contract C {
    function g() public pure returns (bytes4) {
```

(continues on next page)

(continued from previous page)

```
        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:

```
pragma solidity >=0.4.16 <0.7.0;


// 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;
```

(continues on next page)

(continued from previous page)

```
    }

    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:

```
pragma solidity >=0.4.16 <0.7.0;

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 uint(-1);
    }
}

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

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

(continues on next page)

(continued from previous page)

```
    }

    function replace(uint _old, uint _new) public {
        // This performs the library function call
        uint index = data.indexOf(_old);
        if (index == uint(-1))
            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.

2.4.7 Solidity Assembly

Solidity defines an assembly language that you can use without Solidity and also as “inline assembly” inside Solidity source code. This guide starts with describing how to use inline assembly, how it differs from standalone assembly (sometimes also referred to by its proper name “Yul”), and specifies assembly itself.

Inline Assembly

You can interleave Solidity statements with inline assembly in a language close to the one of the virtual machine. This gives you more fine-grained control, especially when you are enhancing the language by writing libraries.

As the EVM is a stack machine, it is often hard to address the correct stack slot and provide arguments to opcodes at the correct point on the stack. Solidity’s inline assembly helps you do this, and with other issues that arise when writing manual assembly.

For inline assembly, the stack is actually not visible at all, but if you look closer, there is always a very direct translation from inline assembly to the stack based EVM opcode stream.

Inline assembly has the following features:

- functional-style opcodes: `mul(1, add(2, 3))`
- assembly-local variables: `let x := add(2, 3) let y := mload(0x40) x := add(x, y)`
- access to external variables: `function f(uint x) public { assembly { x := sub(x, 1) } }`
- loops: `for { let i := 0 } lt(i, x) { i := add(i, 1) } { y := mul(2, y) }`
- if statements: `if slt(x, 0) { x := sub(0, x) }`
- switch statements: `switch x case 0 { y := mul(x, 2) } default { y := 0 }`
- function calls: `function f(x) -> y { switch x case 0 { y := 1 } default { y := mul(x, f(sub(x, 1))) } }`

Warning: Inline assembly is a way to access the Ethereum Virtual Machine at a low level. This bypasses several important safety features and checks of Solidity. You should only use it for tasks that need it, and only if you are confident with using it.

Syntax

Assembly parses comments, literals and identifiers in the same way as Solidity, so you can use the usual `//` and `/* */` comments. There is one exception: Identifiers in inline assembly can contain `.`. Inline assembly is marked by `assembly { ... }` and inside these curly braces, you can use the following (see the later sections for more details):

- literals, i.e. `0x123`, `42` or `"abc"` (strings up to 32 characters)
- opcodes in functional style, e.g. `add(1, mload(0))`
- variable declarations, e.g. `let x := 7`, `let x := add(y, 3)` or `let x` (initial value of 0 is assigned)
- identifiers (assembly-local variables and externals if used as inline assembly), e.g. `add(3, x)`, `sstore(x_slot, 2)`
- assignments, e.g. `x := add(y, 3)`
- blocks where local variables are scoped inside, e.g. `{ let x := 3 { let y := add(x, 1) } }`

Inline assembly manages local variables and control-flow. Because of that, opcodes that interfere with these features are not available. This includes the `dup` and `swap` instructions as well as `jump` instructions and labels.

Example

The following example provides library code to access the code of another contract and load it into a `bytes` variable. This is not possible with "plain Solidity" and the idea is that reusable assembly libraries can enhance the Solidity language without a compiler change.

```
pragma solidity >=0.4.0 <0.7.0;

library GetCode {
    function at(address _addr) public view returns (bytes memory o_code) {
        assembly {
            // retrieve the size of the code, this needs assembly
            let size := extcodesize(_addr)
            // allocate output byte array - this could also be done without assembly
            // by using o_code = new bytes(size)
            o_code := mload(0x40)
            // new "memory end" including padding
            mstore(0x40, add(o_code, and(add(add(size, 0x20), 0x1f), not(0x1f))))
            // store length in memory
            mstore(o_code, size)
            // actually retrieve the code, this needs assembly
            extcodecopy(_addr, add(o_code, 0x20), 0, size)
        }
    }
}
```

Inline assembly is also beneficial in cases where the optimizer fails to produce efficient code, for example:

```
pragma solidity >=0.4.16 <0.7.0;


library VectorSum {
    // This function is less efficient because the optimizer currently fails to
    // remove the bounds checks in array access.
    function sumSolidity(uint[] memory _data) public pure returns (uint sum) {
```

(continues on next page)

(continued from previous page)

```
        for (uint i = 0; i < _data.length; ++i)
            sum += _data[i];
    }

    // We know that we only access the array in bounds, so we can avoid the check.
    // 0x20 needs to be added to an array because the first slot contains the
    // array length.
    function sumAsm(uint[] memory _data) public pure returns (uint sum) {
        for (uint i = 0; i < _data.length; ++i) {
            assembly {
                sum := add(sum, mload(add(add(_data, 0x20), mul(i, 0x20))))
            }
        }
    }

    // Same as above, but accomplish the entire code within inline assembly.
    function sumPureAsm(uint[] memory _data) public pure returns (uint sum) {
        assembly {
            // Load the length (first 32 bytes)
            let len := mload(_data)

            // Skip over the length field.
            //
            // Keep temporary variable so it can be incremented in place.
            //
            // NOTE: incrementing _data would result in an unusable
            //       _data variable after this assembly block
            let data := add(_data, 0x20)

            // Iterate until the bound is not met.
            for
                { let end := add(data, mul(len, 0x20)) }
                lt(data, end)
                { data := add(data, 0x20) }
            {
                sum := add(sum, mload(data))
            }
        }
    }
}
```

Opcodes

This document does not want to be a full description of the Ethereum virtual machine, but the following list can be used as a quick reference of its opcodes.

If an opcode takes arguments, they are given in parentheses. Opcodes marked with - do not return a result, those marked with * are special in a certain way and all others return exactly one value. Opcodes marked with F, H, B, C or I are present since Frontier, Homestead, Byzantium, Constantinople or Istanbul, respectively.

In the following, mem[a...b) signifies the bytes of memory starting at position a up to but not including position b and storage[p] signifies the storage contents at slot p.

In the grammar, opcodes are represented as pre-defined identifiers ("built-in functions").

Instruction			Explanation
stop()	-	F	stop execution, identical to return(0, 0)
add(x, y)		F	x + y
sub(x, y)		F	x - y
mul(x, y)		F	x * y
div(x, y)		F	x / y
sdiv(x, y)		F	x / y, for signed numbers in two's complement
mod(x, y)		F	x % y
smod(x, y)		F	x % y, for signed numbers in two's complement
exp(x, y)		F	x to the power of y
not(x)		F	~x, every bit of x is negated
lt(x, y)		F	1 if x < y, 0 otherwise
gt(x, y)		F	1 if x > y, 0 otherwise
slt(x, y)		F	1 if x < y, 0 otherwise, for signed numbers in two's complement
sgt(x, y)		F	1 if x > y, 0 otherwise, for signed numbers in two's complement
eq(x, y)		F	1 if x == y, 0 otherwise
iszero(x)		F	1 if x == 0, 0 otherwise
and(x, y)		F	bitwise "and" of x and y
or(x, y)		F	bitwise "or" of x and y
xor(x, y)		F	bitwise "xor" of x and y
byte(n, x)		F	nth byte of x, where the most significant byte is the 0th byte
shl(x, y)		C	logical shift left y by x bits
shr(x, y)		C	logical shift right y by x bits
sar(x, y)		C	signed arithmetic shift right y by x bits
addmod(x, y, m)		F	(x + y) % m with arbitrary precision arithmetic
mulmod(x, y, m)		F	(x * y) % m with arbitrary precision arithmetic
signextend(i, x)		F	sign extend from (i*8+7)th bit counting from least significant
keccak256(p, n)		F	keccak(mem[p. . . (p+n)))
pc()		F	current position in code
pop(x)	-	F	discard value x
mload(p)		F	mem[p. . . (p+32))
mstore(p, v)	-	F	mem[p. . . (p+32)) := v
mstore8(p, v)	-	F	mem[p] := v & 0xff (only modifies a single byte)
sload(p)		F	storage[p]
sstore(p, v)	-	F	storage[p] := v
msize()		F	size of memory, i.e. largest accessed memory index
gas()		F	gas still available to execution
address()		F	address of the current contract / execution context
balance(a)		F	wei balance at address a
selfbalance()		I	equivalent to balance(address()), but cheaper
caller()		F	call sender (excluding `delegatecall`)
callvalue()		F	wei sent together with the current call
calldataload(p)		F	call data starting from position p (32 bytes)
calldatasize()		F	size of call data in bytes
calldatacopy(t, f, s)	-	F	copy s bytes from calldata at position f to mem at position t
codesize()		F	size of the code of the current contract / execution context
codecopy(t, f, s)	-	F	copy s bytes from code at position f to mem at position t
extcodesize(a)		F	size of the code at address a
extcodecopy(a, t, f, s)	-	F	like codecopy(t, f, s) but take code at address a
returndatasize()		B	size of the last returndata

Instruction			Explanation
returndatacopy(t, f, s)	-	B	copy s bytes from returndata at position f to mem at position t
extcodehash(a)		C	code hash of address a
create(v, p, n)		F	create new contract with code mem[p. . . (p+n)) and send v wei and return the new
create2(v, p, n, s)		C	create new contract with code mem[p. . . (p+n)) at address keccak256(0xff . this . s
call(g, a, v, in, insize, out, outsize)		F	call contract at address a with input mem[in. . . (in+insize)) providing g gas and v w
callcode(g, a, v, in, insize, out, outsize)		F	identical to `call` but only use the code from a and stay in the context of the curre
delegatecall(g, a, in, insize, out, outsize)		H	identical to `callcode` but also keep `caller` and `callvalue`
staticcall(g, a, in, insize, out, outsize)		B	identical to `call(g, a, 0, in, insize, out, outsize)` but do not
return(p, s)	-	F	end execution, return data mem[p. . . (p+s))
revert(p, s)	-	B	end execution, revert state changes, return data mem[p. . . (p+s))
selfdestruct(a)	-	F	end execution, destroy current contract and send funds to a
invalid()	-	F	end execution with invalid instruction
log0(p, s)	-	F	log without topics and data mem[p. . . (p+s))
log1(p, s, t1)	-	F	log with topic t1 and data mem[p. . . (p+s))
log2(p, s, t1, t2)	-	F	log with topics t1, t2 and data mem[p. . . (p+s))
log3(p, s, t1, t2, t3)	-	F	log with topics t1, t2, t3 and data mem[p. . . (p+s))
log4(p, s, t1, t2, t3, t4)	-	F	log with topics t1, t2, t3, t4 and data mem[p. . . (p+s))
chainid()		I	ID of the executing chain (EIP 1344)
origin()		F	transaction sender
gasprice()		F	gas price of the transaction
blockhash(b)		F	hash of block nr b - only for last 256 blocks excluding current
coinbase()		F	current mining beneficiary
timestamp()		F	timestamp of the current block in seconds since the epoch
number()		F	current block number
difficulty()		F	difficulty of the current block
gaslimit()		F	block gas limit of the current block

Literals

You can use integer constants by typing them in decimal or hexadecimal notation and an appropriate `PUSHi` instruction will automatically be generated. The following creates code to add 2 and 3 resulting in 5 and then computes the bitwise `AND` with the string "abc". The final value is assigned to a local variable called `x`. Strings are stored left-aligned and cannot be longer than 32 bytes.

```
assembly { let x := and("abc", add(3, 2)) }
```

Functional Style

For a sequence of opcodes, it is often hard to see what the actual arguments for certain opcodes are. In the following example, `3` is added to the contents in memory at position `0x80`.

```
3 0x80 mload add 0x80 mstore
```

Solidity inline assembly has a "functional style" notation where the same code would be written as follows:

```
mstore(0x80, add(mload(0x80), 3))
```

If you read the code from right to left, you end up with exactly the same sequence of constants and opcodes, but it is much clearer where the values end up.

If you care about the exact stack layout, just note that the syntactically first argument for a function or opcode will be put at the top of the stack.

Access to External Variables, Functions and Libraries

You can access Solidity variables and other identifiers by using their name. For variables stored in the memory data location, this pushes the address, and not the value onto the stack. Variables stored in the storage data location are different, as they might not occupy a full storage slot, so their "address" is composed of a slot and a byte-offset inside that slot. To retrieve the slot pointed to by the variable `x`, you use `x_slot`, and to retrieve the byte-offset you use `x_offset`.

Local Solidity variables are available for assignments, for example:

```
pragma solidity >=0.4.11 <0.7.0;

contract C {
    uint b;
    function f(uint x) public view returns (uint r) {
        assembly {
            r := mul(x, sload(b_slot)) // ignore the offset, we know it is zero
        }
    }
}
```

Warning: If you access variables of a type that spans less than 256 bits (for example `uint64`, `address`, `bytes16` or `byte`), you cannot make any assumptions about bits not part of the encoding of the type. Especially, do not assume them to be zero. To be safe, always clear the data properly before you use it in a context where this is important: `uint32 x = f(); assembly { x := and(x, 0xffffffff) /* now use x */ }` To clean signed types, you can use the `signextend` opcode: `assembly { signextend(<num_bytes_of_x_minus_one>, x) }`

Declaring Assembly-Local Variables

You can use the `let` keyword to declare variables that are only visible in inline assembly and actually only in the current `{...}`-block. What happens is that the `let` instruction will create a new stack slot that is reserved for the variable and automatically removed again when the end of the block is reached. You need to provide an initial value for the variable which can be just `0`, but it can also be a complex functional-style expression.

Since 0.6.0 the name of a declared variable may not end in `_offset` or `_slot` and it may not shadow any declaration visible in the scope of the inline assembly block (including variable, contract and function declarations). Similarly, if the name of a declared variable contains a dot `.`, the prefix up to the `.` may not conflict with any declaration visible in the scope of the inline assembly block.

```
pragma solidity >=0.4.16 <0.7.0;

contract C {
    function f(uint x) public view returns (uint b) {
        assembly {
            let v := add(x, 1)
            mstore(0x80, v)
            {
                let y := add(sload(v), 1)
```

(continues on next page)

(continued from previous page)

```
                b := y
            } // y is "deallocated" here
            b := add(b, v)
        } // v is "deallocated" here
    }
}
```

Assignments

Assignments are possible to assembly-local variables and to function-local variables. Take care that when you assign to variables that point to memory or storage, you will only change the pointer and not the data.

Variables can only be assigned expressions that result in exactly one value. If you want to assign the values returned from a function that has multiple return parameters, you have to provide multiple variables.

```
{
    let v := 0
    let g := add(v, 2)
    function f() -> a, b { }
    let c, d := f()
}
```

If

The if statement can be used for conditionally executing code. There is no “else” part, consider using “switch” (see below) if you need multiple alternatives.

```
{
    if eq(value, 0) { revert(0, 0) }
}
```

The curly braces for the body are required.

Switch

You can use a switch statement as a very basic version of “if/else”. It takes the value of an expression and compares it to several constants. The branch corresponding to the matching constant is taken. Contrary to the error-prone behaviour of some programming languages, control flow does not continue from one case to the next. There can be a fallback or default case called `default`.

```
{
    let x := 0
    switch calldataload(4)
    case 0 {
        x := calldataload(0x24)
    }
    default {
        x := calldataload(0x44)
    }
    sstore(0, div(x, 2))
}
```

The list of cases does not require curly braces, but the body of a case does require them.

Loops

Assembly supports a simple for-style loop. For-style loops have a header containing an initializing part, a condition and a post-iteration part. The condition has to be a functional-style expression, while the other two are blocks. If the initializing part declares any variables, the scope of these variables is extended into the body (including the condition and the post-iteration part).

The `break` and `continue` statements can be used to exit the loop or skip to the post-part, respectively.

The following example computes the sum of an area in memory.

```
{
    let x := 0
    for { let i := 0 } lt(i, 0x100) { i := add(i, 0x20) } {
        x := add(x, mload(i))
    }
}
```

For loops can also be written so that they behave like while loops: Simply leave the initialization and post-iteration parts empty.

```
{
    let x := 0
    let i := 0
    for { } lt(i, 0x100) { } {      // while(i < 0x100)
        x := add(x, mload(i))
        i := add(i, 0x20)
    }
}
```

Functions

Assembly allows the definition of low-level functions. These take their arguments (and a return PC) from the stack and also put the results onto the stack. Calling a function looks the same way as executing a functional-style opcode.

Functions can be defined anywhere and are visible in the block they are declared in. Inside a function, you cannot access local variables defined outside of that function.

If you call a function that returns multiple values, you have to assign them to a tuple using `a, b := f(x)` or `let a, b := f(x)`.

The `leave` statement can be used to exit the current function. It works like the `return` statement in other languages just that it does not take a value to return, it just exits the functions and the function will return whatever values are currently assigned to the return variable(s).

The following example implements the power function by square-and-multiply.

```
{
    function power(base, exponent) -> result {
        switch exponent
        case 0 { result := 1 }
        case 1 { result := base }
        default {
            result := power(mul(base, base), div(exponent, 2))
```

(continues on next page)

(continued from previous page)

```
            switch mod(exponent, 2)
                case 1 { result := mul(base, result) }
        }
    }
}
```

Things to Avoid

Inline assembly might have a quite high-level look, but it actually is extremely low-level. Function calls, loops, ifs and switches are converted by simple rewriting rules and after that, the only thing the assembler does for you is re-arranging functional-style opcodes, counting stack height for variable access and removing stack slots for assembly-local variables when the end of their block is reached.

Conventions in Solidity

In contrast to EVM assembly, Solidity has types which are narrower than 256 bits, e.g. `uint24`. For efficiency, most arithmetic operations ignore the fact that types can be shorter than 256 bits, and the higher-order bits are cleaned when necessary, i.e., shortly before they are written to memory or before comparisons are performed. This means that if you access such a variable from within inline assembly, you might have to manually clean the higher-order bits first.

Solidity manages memory in the following way. There is a “free memory pointer” at position `0x40` in memory. If you want to allocate memory, use the memory starting from where this pointer points at and update it. There is no guarantee that the memory has not been used before and thus you cannot assume that its contents are zero bytes. There is no built-in mechanism to release or free allocated memory. Here is an assembly snippet you can use for allocating memory that follows the process outlined above:

```
function allocate(length) -> pos {
  pos := mload(0x40)
  mstore(0x40, add(pos, length))
}
```

The first 64 bytes of memory can be used as “scratch space” for short-term allocation. The 32 bytes after the free memory pointer (i.e., starting at `0x60`) are meant to be zero permanently and is used as the initial value for empty dynamic memory arrays. This means that the allocatable memory starts at `0x80`, which is the initial value of the free memory pointer.

Elements in memory arrays in Solidity always occupy multiples of 32 bytes (this is even true for `byte[]`, but not for `bytes` and `string`). Multi-dimensional memory arrays are pointers to memory arrays. The length of a dynamic array is stored at the first slot of the array and followed by the array elements.

> **Warning:** Statically-sized memory arrays do not have a length field, but it might be added later to allow better convertibility between statically- and dynamically-sized arrays, so do not rely on this.

Standalone Assembly

The assembly language described as inline assembly above can also be used standalone and in fact, the plan is to use it as an intermediate language for the Solidity compiler. In this form, it tries to achieve several goals:

1. Programs written in it should be readable, even if the code is generated by a compiler from Solidity.
2. The translation from assembly to bytecode should contain as few “surprises” as possible.

3. Control flow should be easy to detect to help in formal verification and optimization.

In order to achieve the first and last goal, assembly provides high-level constructs like `for` loops, `if` and `switch` statements and function calls. It should be possible to write assembly programs that do not make use of explicit `SWAP`, `DUP`, `JUMP` and `JUMPI` statements, because the first two obfuscate the data flow and the last two obfuscate control flow. Furthermore, functional statements of the form `mul(add(x, y), 7)` are preferred over pure opcode statements like `7 y x add mul` because in the first form, it is much easier to see which operand is used for which opcode.

The second goal is achieved by compiling the higher level constructs to bytecode in a very regular way. The only non-local operation performed by the assembler is name lookup of user-defined identifiers (functions, variables, . . .), which follow very simple and regular scoping rules and cleanup of local variables from the stack.

Scoping: An identifier that is declared (label, variable, function, assembly) is only visible in the block where it was declared (including nested blocks inside the current block). It is not legal to access local variables across function borders, even if they would be in scope. Shadowing is not allowed. Local variables cannot be accessed before they were declared, but functions and assemblies can. Assemblies are special blocks that are used for e.g. returning runtime code or creating contracts. No identifier from an outer assembly is visible in a sub-assembly.

If control flow passes over the end of a block, pop instructions are inserted that match the number of local variables declared in that block. Whenever a local variable is referenced, the code generator needs to know its current relative position in the stack and thus it needs to keep track of the current so-called stack height. Since all local variables are removed at the end of a block, the stack height before and after the block should be the same. If this is not the case, compilation fails.

Using `switch`, `for` and functions, it should be possible to write complex code without using `jump` or `jumpi` manually. This makes it much easier to analyze the control flow, which allows for improved formal verification and optimization.

Furthermore, if manual jumps are allowed, computing the stack height is rather complicated. The position of all local variables on the stack needs to be known, otherwise neither references to local variables nor removing local variables automatically from the stack at the end of a block will work properly.

Example:

We will follow an example compilation from Solidity to assembly. We consider the runtime bytecode of the following Solidity program:

```
pragma solidity >=0.4.16 <0.7.0;


contract C {
    function f(uint x) public pure returns (uint y) {
        y = 1;
        for (uint i = 0; i < x; i++)
            y = 2 * y;
    }
}
```

The following assembly will be generated:

```
{
  mstore(0x40, 0x80) // store the "free memory pointer"
  // function dispatcher
  switch div(calldataload(0), exp(2, 226))
  case 0xb3de648b {
    let r := f(calldataload(4))
    let ret := $allocate(0x20)
    mstore(ret, r)
```

(continues on next page)

(continued from previous page)

```
    return(ret, 0x20)
  }
  default { revert(0, 0) }
  // memory allocator
  function $allocate(size) -> pos {
    pos := mload(0x40)
    mstore(0x40, add(pos, size))
  }
  // the contract function
  function f(x) -> y {
    y := 1
    for { let i := 0 } lt(i, x) { i := add(i, 1) } {
      y := mul(2, y)
    }
  }
}
```

Assembly Grammar

The tasks of the parser are the following:

- Turn the byte stream into a token stream, discarding C++-style comments (a special comment exists for source references, but we will not explain it here).
- Turn the token stream into an AST according to the grammar below
- Register identifiers with the block they are defined in (annotation to the AST node) and note from which point on, variables can be accessed.

The assembly lexer follows the one defined by Solidity itself.

Whitespace is used to delimit tokens and it consists of the characters Space, Tab and Linefeed. Comments are regular JavaScript/C++ comments and are interpreted in the same way as Whitespace.

Grammar:

```
AssemblyBlock = '{' AssemblyItem* '}'
AssemblyItem =
    Identifier |
    AssemblyBlock |
    AssemblyExpression |
    AssemblyLocalDefinition |
    AssemblyAssignment |
    AssemblyIf |
    AssemblySwitch |
    AssemblyFunctionDefinition |
    AssemblyFor |
    'break' |
    'continue' |
    'leave' |
    SubAssembly
AssemblyExpression = AssemblyCall | Identifier | AssemblyLiteral
AssemblyLiteral = NumberLiteral | StringLiteral | HexLiteral
Identifier = [a-zA-Z_$] [a-zA-Z_0-9.]*
AssemblyCall = Identifier '(' ( AssemblyExpression ( ',' AssemblyExpression )* )? ')'
AssemblyLocalDefinition = 'let' IdentifierOrList ( ':=' AssemblyExpression )?
AssemblyAssignment = IdentifierOrList ':=' AssemblyExpression
```

(continues on next page)

(continued from previous page)

```
IdentifierOrList = Identifier | '(' IdentifierList ')'
IdentifierList = Identifier ( ',' Identifier)*
AssemblyIf = 'if' AssemblyExpression AssemblyBlock
AssemblySwitch = 'switch' AssemblyExpression AssemblyCase*
    ( 'default' AssemblyBlock )?
AssemblyCase = 'case' AssemblyExpression AssemblyBlock
AssemblyFunctionDefinition = 'function' Identifier '(' IdentifierList? ')'
    ( '->' '(' IdentifierList ')' )? AssemblyBlock
AssemblyFor = 'for' ( AssemblyBlock | AssemblyExpression )
    AssemblyExpression ( AssemblyBlock | AssemblyExpression ) AssemblyBlock
SubAssembly = 'assembly' Identifier AssemblyBlock
NumberLiteral = HexNumber | DecimalNumber
HexLiteral = 'hex' ('"' ([0-9a-fA-F]{2})* '"' | '\'' ([0-9a-fA-F]{2})* '\'')
StringLiteral = '"' ([^"\r\n\\] | '\\' .)* '"'
HexNumber = '0x' [0-9a-fA-F]+
DecimalNumber = [0-9]+
```

2.4.8 Miscellaneous

Layout of State Variables in Storage

Statically-sized variables (everything except mapping and dynamically-sized array types) are laid out contiguously in storage starting from position `0`. Multiple, contiguous items that need less than 32 bytes are packed into a single storage slot if possible, according to the following rules:

- The first item in a storage slot is stored lower-order aligned.
- Elementary types use only as many bytes as are necessary to store them.
- If an elementary type does not fit the remaining part of a storage slot, it is moved to the next storage slot.
- Structs and array data always start a new slot and occupy whole slots (but items inside a struct or array are packed tightly according to these rules).

For contracts that use inheritance, the ordering of state variables is determined by the C3-linearized order of contracts starting with the most base-ward contract. If allowed by the above rules, state variables from different contracts do share the same storage slot.

The elements of structs and arrays are stored after each other, just as if they were given explicitly.

Warning: When using elements that are smaller than 32 bytes, your contract's gas usage may be higher. This is because the EVM operates on 32 bytes at a time. Therefore, if the element is smaller than that, the EVM must use more operations in order to reduce the size of the element from 32 bytes to the desired size.

It is only beneficial to use reduced-size arguments if you are dealing with storage values because the compiler will pack multiple elements into one storage slot, and thus, combine multiple reads or writes into a single operation. When dealing with function arguments or memory values, there is no inherent benefit because the compiler does not pack these values.

Finally, in order to allow the EVM to optimize for this, ensure that you try to order your storage variables and `struct` members such that they can be packed tightly. For example, declaring your storage variables in the order of `uint128, uint128, uint256` instead of `uint128, uint256, uint128`, as the former will only take up two slots of storage whereas the latter will take up three.

Note: The layout of state variables in storage is considered to be part of the external interface of Solidity due to the fact that storage pointers can be passed to libraries. This means that any change to the rules outlined in this section is considered a breaking change of the language and due to its critical nature should be considered very carefully before being executed.

Mappings and Dynamic Arrays

Due to their unpredictable size, mapping and dynamically-sized array types use a Keccak-256 hash computation to find the starting position of the value or the array data. These starting positions are always full stack slots.

The mapping or the dynamic array itself occupies a slot in storage at some position `p` according to the above rule (or by recursively applying this rule for mappings of mappings or arrays of arrays). For dynamic arrays, this slot stores the number of elements in the array (byte arrays and strings are an exception, see *below*). For mappings, the slot is unused (but it is needed so that two equal mappings after each other will use a different hash distribution). Array data is located at `keccak256(p)` and the value corresponding to a mapping key `k` is located at `keccak256(k . p)` where `.` is concatenation. If the value is again a non-elementary type, the positions are found by adding an offset of `keccak256(k . p)`.

So for the following contract snippet the position of `data[4][9].b` is at `keccak256(uint256(9) . keccak256(uint256(4) . uint256(1))) + 1`:

```
pragma solidity >=0.4.0 <0.7.0;


contract C {
    struct S { uint a; uint b; }
    uint x;
    mapping(uint => mapping(uint => S)) data;
}
```

`bytes` and `string`

`bytes` and `string` are encoded identically. For short byte arrays, they store their data in the same slot where the length is also stored. In particular: if the data is at most `31` bytes long, it is stored in the higher-order bytes (left aligned) and the lowest-order byte stores `length * 2`. For byte arrays that store data which is `32` or more bytes long, the main slot stores `length * 2 + 1` and the data is stored as usual in `keccak256(slot)`. This means that you can distinguish a short array from a long array by checking if the lowest bit is set: short (not set) and long (set).

Note: Handling invalidly encoded slots is currently not supported but may be added in the future.

JSON Output

The storage layout of a contract can be requested via the *standard JSON interface*. The output is a JSON object containing two keys, `storage` and `types`. The `storage` object is an array where each element has the following form:

```
{
    "astId": 2,
    "contract": "fileA:A",
    "label": "x",
    "offset": 0,
    "slot": "0",
    "type": "t_uint256"
}
```

The example above is the storage layout of `contract A { uint x; }` from source unit `fileA` and

- `astId` is the id of the AST node of the state variable's declaration
- `contract` is the name of the contract including its path as prefix
- `label` is the name of the state variable
- `offset` is the offset in bytes within the storage slot according to the encoding
- `slot` is the storage slot where the state variable resides or starts. This number may be very large and therefore its JSON value is represented as a string.
- `type` is an identifier used as key to the variable's type information (described in the following)

The given `type`, in this case `t_uint256` represents an element in `types`, which has the form:

```
{
    "encoding": "inplace",
    "label": "uint256",
    "numberOfBytes": "32",
}
```

where

- `encoding` how the data is encoded in storage, where the possible values are:
  - `inplace`: data is laid out contiguously in storage (see *above*).
  - `mapping`: Keccak-256 hash-based method (see *above*).
  - `dynamic_array`: Keccak-256 hash-based method (see *above*).
  - `bytes`: single slot or Keccak-256 hash-based depending on the data size (see *above*).
- `label` is the canonical type name.
- `numberOfBytes` is the number of used bytes (as a decimal string). Note that if `numberOfBytes > 32` this means that more than one slot is used.

Some types have extra information besides the four above. Mappings contain its `key` and `value` types (again referencing an entry in this mapping of types), arrays have its `base` type, and structs list their `members` in the same format as the top-level `storage` (see *above*).

Note: The JSON output format of a contract's storage layout is still considered experimental and is subject to change in non-breaking releases of Solidity.

The following example shows a contract and its storage layout, containing value and reference types, types that are encoded packed, and nested types.

```
pragma solidity >=0.4.0 <0.7.0;
contract A {
    struct S {
        uint128 a;
        uint128 b;
        uint[2] staticArray;
        uint[] dynArray;
    }

    uint x;
    uint y;
    S s;
    address addr;
    mapping (uint => mapping (address => bool)) map;
    uint[] array;
    string s1;
    bytes b1;
}
```

```
"storageLayout": {
  "storage": [
    {
      "astId": 14,
      "contract": "fileA:A",
      "label": "x",
      "offset": 0,
      "slot": "0",
      "type": "t_uint256"
    },
    {
      "astId": 16,
      "contract": "fileA:A",
      "label": "y",
      "offset": 0,
      "slot": "1",
      "type": "t_uint256"
    },
    {
      "astId": 18,
      "contract": "fileA:A",
      "label": "s",
      "offset": 0,
      "slot": "2",
      "type": "t_struct(S)12_storage"
    },
    {
      "astId": 20,
      "contract": "fileA:A",
      "label": "addr",
      "offset": 0,
      "slot": "6",
      "type": "t_address"
    },
    {
      "astId": 26,
      "contract": "fileA:A",
      "label": "map",
```

(continues on next page)

(continued from previous page)

```
      "offset": 0,
      "slot": "7",
      "type": "t_mapping(t_uint256,t_mapping(t_address,t_bool))"
    },
    {
      "astId": 29,
      "contract": "fileA:A",
      "label": "array",
      "offset": 0,
      "slot": "8",
      "type": "t_array(t_uint256)dyn_storage"
    },
    {
      "astId": 31,
      "contract": "fileA:A",
      "label": "s1",
      "offset": 0,
      "slot": "9",
      "type": "t_string_storage"
    },
    {
      "astId": 33,
      "contract": "fileA:A",
      "label": "b1",
      "offset": 0,
      "slot": "10",
      "type": "t_bytes_storage"
    }
  ],
  "types": {
    "t_address": {
      "encoding": "inplace",
      "label": "address",
      "numberOfBytes": "20"
    },
    "t_array(t_uint256)2_storage": {
      "base": "t_uint256",
      "encoding": "inplace",
      "label": "uint256[2]",
      "numberOfBytes": "64"
    },
    "t_array(t_uint256)dyn_storage": {
      "base": "t_uint256",
      "encoding": "dynamic_array",
      "label": "uint256[]",
      "numberOfBytes": "32"
    },
    "t_bool": {
      "encoding": "inplace",
      "label": "bool",
      "numberOfBytes": "1"
    },
    "t_bytes_storage": {
      "encoding": "bytes",
      "label": "bytes",
      "numberOfBytes": "32"
    },
```

(continues on next page)

(continued from previous page)

```
    "t_mapping(t_address,t_bool)": {
      "encoding": "mapping",
      "key": "t_address",
      "label": "mapping(address => bool)",
      "numberOfBytes": "32",
      "value": "t_bool"
    },
    "t_mapping(t_uint256,t_mapping(t_address,t_bool))": {
      "encoding": "mapping",
      "key": "t_uint256",
      "label": "mapping(uint256 => mapping(address => bool))",
      "numberOfBytes": "32",
      "value": "t_mapping(t_address,t_bool)"
    },
    "t_string_storage": {
      "encoding": "bytes",
      "label": "string",
      "numberOfBytes": "32"
    },
    "t_struct(S)12_storage": {
      "encoding": "inplace",
      "label": "struct A.S",
      "members": [
        {
          "astId": 2,
          "contract": "fileA:A",
          "label": "a",
          "offset": 0,
          "slot": "0",
          "type": "t_uint128"
        },
        {
          "astId": 4,
          "contract": "fileA:A",
          "label": "b",
          "offset": 16,
          "slot": "0",
          "type": "t_uint128"
        },
        {
          "astId": 8,
          "contract": "fileA:A",
          "label": "staticArray",
          "offset": 0,
          "slot": "1",
          "type": "t_array(t_uint256)2_storage"
        },
        {
          "astId": 11,
          "contract": "fileA:A",
          "label": "dynArray",
          "offset": 0,
          "slot": "3",
          "type": "t_array(t_uint256)dyn_storage"
        }
      ],
      "numberOfBytes": "128"
```

(continues on next page)

(continued from previous page)

```
    },
    "t_uint128": {
      "encoding": "inplace",
      "label": "uint128",
      "numberOfBytes": "16"
    },
    "t_uint256": {
      "encoding": "inplace",
      "label": "uint256",
      "numberOfBytes": "32"
    }
  }
}
```

Layout in Memory

Solidity reserves four 32-byte slots, with specific byte ranges (inclusive of endpoints) being used as follows:

- `0x00` - `0x3f` (64 bytes): scratch space for hashing methods
- `0x40` - `0x5f` (32 bytes): currently allocated memory size (aka. free memory pointer)
- `0x60` - `0x7f` (32 bytes): zero slot

Scratch space can be used between statements (i.e. within inline assembly). The zero slot is used as initial value for dynamic memory arrays and should never be written to (the free memory pointer points to `0x80` initially).

Solidity always places new objects at the free memory pointer and memory is never freed (this might change in the future).

Elements in memory arrays in Solidity always occupy multiples of 32 bytes (this is even true for `byte[]`, but not for `bytes` and `string`). Multi-dimensional memory arrays are pointers to memory arrays. The length of a dynamic array is stored at the first slot of the array and followed by the array elements.

> **Warning:** There are some operations in Solidity that need a temporary memory area larger than 64 bytes and therefore will not fit into the scratch space. They will be placed where the free memory points to, but given their short lifetime, the pointer is not updated. The memory may or may not be zeroed out. Because of this, one should not expect the free memory to point to zeroed out memory.
>
> While it may seem like a good idea to use `msize` to arrive at a definitely zeroed out memory area, using such a pointer non-temporarily without updating the free memory pointer can have unexpected results.

Layout of Call Data

The input data for a function call is assumed to be in the format defined by the *ABI specification*. Among others, the ABI specification requires arguments to be padded to multiples of 32 bytes. The internal function calls use a different convention.

Arguments for the constructor of a contract are directly appended at the end of the contract's code, also in ABI encoding. The constructor will access them through a hard-coded offset, and not by using the `codesize` opcode, since this of course changes when appending data to the code.

Internals - Cleaning Up Variables

When a value is shorter than 256 bit, in some cases the remaining bits must be cleaned. The Solidity compiler is designed to clean such remaining bits before any operations that might be adversely affected by the potential garbage in the remaining bits. For example, before writing a value to memory, the remaining bits need to be cleared because the memory contents can be used for computing hashes or sent as the data of a message call. Similarly, before storing a value in the storage, the remaining bits need to be cleaned because otherwise the garbled value can be observed.

On the other hand, we do not clean the bits if the immediately following operation is not affected. For instance, since any non-zero value is considered `true` by `JUMPI` instruction, we do not clean the boolean values before they are used as the condition for `JUMPI`.

In addition to the design principle above, the Solidity compiler cleans input data when it is loaded onto the stack.

Different types have different rules for cleaning up invalid values:

Type	Valid Values	Invalid Values Mean
enum of n members	0 until n - 1	exception
bool	0 or 1	1
signed integers	sign-extended word	currently silently wraps; in the future exceptions will be thrown
unsigned integers	higher bits zeroed	currently silently wraps; in the future exceptions will be thrown

Internals - The Optimiser

This section discusses the optimiser that was first added to Solidity, which operates on opcode streams. For information on the new Yul-based optimiser, please see the readme on github.

The Solidity optimiser operates on assembly. It splits the sequence of instructions into basic blocks at `JUMPs` and `JUMPDESTs`. Inside these blocks, the optimiser analyses the instructions and records every modification to the stack, memory, or storage as an expression which consists of an instruction and a list of arguments which are pointers to other expressions. The optimiser uses a component called “CommonSubexpressionEliminator” that amongst other tasks, finds expressions that are always equal (on every input) and combines them into an expression class. The optimiser first tries to find each new expression in a list of already known expressions. If this does not work, it simplifies the expression according to rules like `constant + constant = sum_of_constants` or `X * 1 = X`. Since this is a recursive process, we can also apply the latter rule if the second factor is a more complex expression where we know that it always evaluates to one. Modifications to storage and memory locations have to erase knowledge about storage and memory locations which are not known to be different. If we first write to location x and then to location y and both are input variables, the second could overwrite the first, so we do not know what is stored at x after we wrote to y. If simplification of the expression x - y evaluates to a non-zero constant, we know that we can keep our knowledge about what is stored at x.

After this process, we know which expressions have to be on the stack at the end, and have a list of modifications to memory and storage. This information is stored together with the basic blocks and is used to link them. Furthermore, knowledge about the stack, storage and memory configuration is forwarded to the next block(s). If we know the targets of all `JUMP` and `JUMPI` instructions, we can build a complete control flow graph of the program. If there is only one target we do not know (this can happen as in principle, jump targets can be computed from inputs), we have to erase all knowledge about the input state of a block as it can be the target of the unknown `JUMP`. If the optimiser finds a `JUMPI` whose condition evaluates to a constant, it transforms it to an unconditional jump.

As the last step, the code in each block is re-generated. The optimiser creates a dependency graph from the expressions on the stack at the end of the block, and it drops every operation that is not part of this graph. It generates code that applies the modifications to memory and storage in the order they were made in the original code (dropping modifications which were found not to be needed). Finally, it generates all values that are required to be on the stack in the correct place.

These steps are applied to each basic block and the newly generated code is used as replacement if it is smaller. If a basic block is split at a `JUMPI` and during the analysis, the condition evaluates to a constant, the `JUMPI` is replaced depending on the value of the constant. Thus code like

```
uint x = 7;
data[7] = 9;
if (data[x] != x + 2)
  return 2;
else
  return 1;
```

still simplifies to code which you can compile even though the instructions contained a jump in the beginning of the process:

```
data[7] = 9;
return 1;
```

Source Mappings

As part of the AST output, the compiler provides the range of the source code that is represented by the respective node in the AST. This can be used for various purposes ranging from static analysis tools that report errors based on the AST and debugging tools that highlight local variables and their uses.

Furthermore, the compiler can also generate a mapping from the bytecode to the range in the source code that generated the instruction. This is again important for static analysis tools that operate on bytecode level and for displaying the current position in the source code inside a debugger or for breakpoint handling. This mapping also contains other information, like the jump type and the modifier depth (see below).

Both kinds of source mappings use integer identifiers to refer to source files. The identifier of a source file is stored in `output['sources'][sourceName]['id']` where `output` is the output of the standard-json compiler interface parsed as JSON.

Note: In the case of instructions that are not associated with any particular source file, the source mapping assigns an integer identifier of `-1`. This may happen for bytecode sections stemming from compiler-generated inline assembly statements.

The source mappings inside the AST use the following notation:

`s:l:f`

Where `s` is the byte-offset to the start of the range in the source file, `l` is the length of the source range in bytes and `f` is the source index mentioned above.

The encoding in the source mapping for the bytecode is more complicated: It is a list of `s:l:f:j:m` separated by `;`. Each of these elements corresponds to an instruction, i.e. you cannot use the byte offset but have to use the instruction offset (push instructions are longer than a single byte). The fields `s`, `l` and `f` are as above. `j` can be either `i`, `o` or `-` signifying whether a jump instruction goes into a function, returns from a function or is a regular jump as part of e.g. a loop. The last field, `m`, is an integer that denotes the "modifier depth". This depth is increased whenever the placeholder statement (`_`) is entered in a modifier and decreased when it is left again. This allows debuggers to track tricky cases like the same modifier being used twice or multiple placeholder statements being used in a single modifier.

In order to compress these source mappings especially for bytecode, the following rules are used:

- If a field is empty, the value of the preceding element is used.
- If a `:` is missing, all following fields are considered empty.

This means the following source mappings represent the same information:

`1:2:1;1:9:1;2:1:2;2:1:2;2:1:2`

`1:2:1;:9;2:1:2;;`

Tips and Tricks

- Use `delete` on arrays to delete all its elements.
- Use shorter types for struct elements and sort them such that short types are grouped together. This can lower the gas costs as multiple `SSTORE` operations might be combined into a single (`SSTORE` costs 5000 or 20000 gas, so this is what you want to optimise). Use the gas price estimator (with optimiser enabled) to check!
- Make your state variables public - the compiler creates *getters* for you automatically.
- If you end up checking conditions on input or state a lot at the beginning of your functions, try using *Function Modifiers*.
- Initialize storage structs with a single assignment: `x = MyStruct({a: 1, b: 2});`

Note: If the storage struct has tightly packed properties, initialize it with separate assignments: `x.a = 1; x.b = 2;`. In this way it will be easier for the optimizer to update storage in one go, thus making assignment cheaper.

Cheatsheet

Order of Precedence of Operators

The following is the order of precedence for operators, listed in order of evaluation.

Precedence	Description	Operator
1	Postfix increment and decrement	++, --
	New expression	new <typename>
	Array subscripting	<array>[<index>]
	Member access	<object>.<member>
	Function-like call	<func>(<args...>)
	Parentheses	(<statement>)
2	Prefix increment and decrement	++, --
	Unary minus	-
	Unary operations	delete
	Logical NOT	!
	Bitwise NOT	~
3	Exponentiation	**
4	Multiplication, division and modulo	*, /, %
5	Addition and subtraction	+, -
6	Bitwise shift operators	<<, >>
7	Bitwise AND	&
8	Bitwise XOR	^
9	Bitwise OR	|
10	Inequality operators	<, >, <=, >=
11	Equality operators	==, !=
12	Logical AND	&&
13	Logical OR	||
14	Ternary operator	<conditional> ? <if-true> : <if-false>
	Assignment operators	=, |=, ^=, &=, <<=, >>=, +=, -=, *=, /=, %=
15	Comma operator	,

Global Variables

- abi.decode(bytes memory encodedData, (...)) returns (...): *ABI*-decodes the provided data. The types are given in parentheses as second argument. Example: (uint a, uint[2] memory b, bytes memory c) = abi.decode(data, (uint, uint[2], bytes))
- abi.encode(...) returns (bytes memory): *ABI*-encodes the given arguments
- abi.encodePacked(...) returns (bytes memory): Performs *packed encoding* of the given arguments. Note that this encoding can be ambiguous!
- abi.encodeWithSelector(bytes4 selector, ...) returns (bytes memory): *ABI*-encodes the given arguments starting from the second and prepends the given four-byte selector
- abi.encodeWithSignature(string memory signature, ...) returns (bytes memory): Equivalent to abi.encodeWithSelector(bytes4(keccak256(bytes(signature)), ...)`
- block.coinbase (address payable): current block miner's address
- block.difficulty (uint): current block difficulty
- block.gaslimit (uint): current block gaslimit
- block.number (uint): current block number
- block.timestamp (uint): current block timestamp
- gasleft() returns (uint256): remaining gas

- `msg.data` (`bytes`): complete calldata
- `msg.sender` (`address payable`): sender of the message (current call)
- `msg.value` (`uint`): number of wei sent with the message
- `now` (`uint`): current block timestamp (alias for `block.timestamp`)
- `tx.gasprice` (`uint`): gas price of the transaction
- `tx.origin` (`address payable`): sender of the transaction (full call chain)
- `assert(bool condition)`: abort execution and revert state changes if condition is `false` (use for internal error)
- `require(bool condition)`: abort execution and revert state changes if condition is `false` (use for malformed input or error in external component)
- `require(bool condition, string memory message)`: abort execution and revert state changes if condition is `false` (use for malformed input or error in external component). Also provide error message.
- `revert()`: abort execution and revert state changes
- `revert(string memory message)`: abort execution and revert state changes providing an explanatory string
- `blockhash(uint blockNumber) returns (bytes32)`: hash of the given block - only works for 256 most recent blocks
- `keccak256(bytes memory) returns (bytes32)`: compute the Keccak-256 hash of the input
- `sha256(bytes memory) returns (bytes32)`: compute the SHA-256 hash of the input
- `ripemd160(bytes memory) returns (bytes20)`: compute the RIPEMD-160 hash of the input
- `ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)`: recover address associated with the public key from elliptic curve signature, return zero on error
- `addmod(uint x, uint y, uint k) returns (uint)`: compute `(x + y) % k` where the addition is performed with arbitrary precision and does not wrap around at `2**256`. Assert that `k != 0` starting from version 0.5.0.
- `mulmod(uint x, uint y, uint k) returns (uint)`: compute `(x * y) % k` where the multiplication is performed with arbitrary precision and does not wrap around at `2**256`. Assert that `k != 0` starting from version 0.5.0.
- `this` (current contract's type): the current contract, explicitly convertible to `address` or `address payable`
- `super`: the contract one level higher in the inheritance hierarchy
- `selfdestruct(address payable recipient)`: destroy the current contract, sending its funds to the given address
- `<address>.balance` (`uint256`): balance of the *Address* in Wei
- `<address payable>.send(uint256 amount) returns (bool)`: send given amount of Wei to *Address*, returns `false` on failure
- `<address payable>.transfer(uint256 amount)`: send given amount of Wei to *Address*, throws on failure
- `type(C).name` (`string`): the name of the contract
- `type(C).creationCode` (`bytes memory`): creation bytecode of the given contract, see *Type Information*.

- `type(C).runtimeCode (bytes memory)`: runtime bytecode of the given contract, see *Type Information*.

Note: Do not rely on `block.timestamp`, `now` and `blockhash` as a source of randomness, unless you know what you are doing.

Both the timestamp and the block hash can be influenced by miners to some degree. Bad actors in the mining community can for example run a casino payout function on a chosen hash and just retry a different hash if they did not receive any money.

The current block timestamp must be strictly larger than the timestamp of the last block, but the only guarantee is that it will be somewhere between the timestamps of two consecutive blocks in the canonical chain.

Note: The block hashes are not available for all blocks for scalability reasons. You can only access the hashes of the most recent 256 blocks, all other values will be zero.

Note: In version 0.5.0, the following aliases were removed: `suicide` as alias for `selfdestruct`, `msg.gas` as alias for `gasleft`, `block.blockhash` as alias for `blockhash` and `sha3` as alias for `keccak256`.

Function Visibility Specifiers

```
function myFunction() <visibility specifier> returns (bool) {
    return true;
}
```

- `public`: visible externally and internally (creates a *getter function* for storage/state variables)
- `private`: only visible in the current contract
- `external`: only visible externally (only for functions) - i.e. can only be message-called (via `this.func`)
- `internal`: only visible internally

Modifiers

- `pure` for functions: Disallows modification or access of state.
- `view` for functions: Disallows modification of state.
- `payable` for functions: Allows them to receive Ether together with a call.
- `constant` for state variables: Disallows assignment (except initialisation), does not occupy storage slot.
- `anonymous` for events: Does not store event signature as topic.
- `indexed` for event parameters: Stores the parameter as topic.
- `virtual` for functions and modifiers: Allows the function's or modifier's behaviour to be changed in derived contracts.
- `override`: States that this function, modifier or public state variable changes the behaviour of a function or modifier in a base contract.

Reserved Keywords

These keywords are reserved in Solidity. They might become part of the syntax in the future:

`after`, `alias`, `apply`, `auto`, `case`, `copyof`, `default`, `define`, `final`, `immutable`, `implements`, `in`, `inline`, `let`, `macro`, `match`, `mutable`, `null`, `of`, `partial`, `promise`, `reference`, `relocatable`, `sealed`, `sizeof`, `static`, `supports`, `switch`, `typedef`, `typeof`, `unchecked`.

Language Grammar

```
SourceUnit = (PragmaDirective | ImportDirective | ContractDefinition)*

// Pragma actually parses anything up to the trailing ';' to be fully forward-
↪compatible.
PragmaDirective = 'pragma' Identifier ([^;]+) ';'

ImportDirective = 'import' StringLiteral ('as' Identifier)? ';'
        | 'import' ('*' | Identifier) ('as' Identifier)? 'from' StringLiteral ';'
        | 'import' '{' Identifier ('as' Identifier)? ( ',' Identifier ('as'␣
↪Identifier)? )* '}' 'from' StringLiteral ';'

ContractDefinition = 'abstract'? ( 'contract' | 'library' | 'interface' ) Identifier
                     ( 'is' InheritanceSpecifier (',' InheritanceSpecifier )* )?
                     '{' ContractPart* '}'

ContractPart = StateVariableDeclaration | UsingForDeclaration
             | StructDefinition | ModifierDefinition | FunctionDefinition |␣
↪EventDefinition | EnumDefinition

InheritanceSpecifier = UserDefinedTypeName ( '(' Expression ( ',' Expression )* ')' )?

StateVariableDeclaration = TypeName ( 'public' | 'internal' | 'private' | 'constant'␣
↪| OverrideSpecifier )* Identifier ('=' Expression)? ';'
UsingForDeclaration = 'using' Identifier 'for' ('*' | TypeName) ';'
StructDefinition = 'struct' Identifier '{'
                     ( VariableDeclaration ';' (VariableDeclaration ';')* ) '}'

ModifierDefinition = 'modifier' Identifier ParameterList?  ( 'virtual' |␣
↪OverrideSpecifier )* Block
ModifierInvocation = Identifier ( '(' ExpressionList? ')' )?

FunctionDefinition = 'function' Identifier? ParameterList
                     ( ModifierInvocation | StateMutability | 'external' | 'public' |
↪'internal' | 'private' | 'virtual' | OverrideSpecifier )*
                     ( 'returns' ParameterList )? ( ';' | Block )

OverrideSpecifier = 'override' ( '(' UserDefinedTypeName (',' UserDefinedTypeName)* ')
↪' )?

EventDefinition = 'event' Identifier EventParameterList 'anonymous'? ';'

EnumValue = Identifier
EnumDefinition = 'enum' Identifier '{' EnumValue? (',' EnumValue)* '}'

ParameterList = '(' ( Parameter (',' Parameter)* )? ')'
Parameter = TypeName StorageLocation? Identifier?
```

(continues on next page)

(continued from previous page)

```
EventParameterList = '(' ( EventParameter (',' EventParameter )* )? ')'
EventParameter = TypeName 'indexed'? Identifier?

FunctionTypeParameterList = '(' ( FunctionTypeParameter (',' FunctionTypeParameter )*␣
↪)? ')'
FunctionTypeParameter = TypeName StorageLocation?

// semantic restriction: mappings and structs (recursively) containing mappings
// are not allowed in argument lists
VariableDeclaration = TypeName StorageLocation? Identifier

TypeName = ElementaryTypeName
         | UserDefinedTypeName
         | Mapping
         | ArrayTypeName
         | FunctionTypeName
         | ( 'address' 'payable' )

UserDefinedTypeName = Identifier ( '.' Identifier )*

Mapping = 'mapping' '(' ElementaryTypeName '=>' TypeName ')'
ArrayTypeName = TypeName '[' Expression? ']'
FunctionTypeName = 'function' FunctionTypeParameterList ( 'internal' | 'external' |␣
↪StateMutability )*
                   ( 'returns' FunctionTypeParameterList )?
StorageLocation = 'memory' | 'storage' | 'calldata'
StateMutability = 'pure' | 'view' | 'payable'

Block = '{' Statement* '}'
Statement = IfStatement | TryStatement | WhileStatement | ForStatement | Block |␣
↪InlineAssemblyStatement |
            ( DoWhileStatement | PlaceholderStatement | Continue | Break | Return |
              Throw | EmitStatement | SimpleStatement ) ';'

ExpressionStatement = Expression
IfStatement = 'if' '(' Expression ')' Statement ( 'else' Statement )?
TryStatement = 'try' Expression ( 'returns' ParameterList )? Block CatchClause+
CatchClause = 'catch' Identifier? ParameterList Block
WhileStatement = 'while' '(' Expression ')' Statement
PlaceholderStatement = '_'
SimpleStatement = VariableDefinition | ExpressionStatement
ForStatement = 'for' '(' (SimpleStatement)? ';' (Expression)? ';'␣
↪(ExpressionStatement)? ')' Statement
InlineAssemblyStatement = 'assembly' StringLiteral? AssemblyBlock
DoWhileStatement = 'do' Statement 'while' '(' Expression ')'
Continue = 'continue'
Break = 'break'
Return = 'return' Expression?
Throw = 'throw'
EmitStatement = 'emit' FunctionCall
VariableDefinition = (VariableDeclaration | '(' VariableDeclaration? (','␣
↪VariableDeclaration? )* ')' ) ( '=' Expression )?

// Precedence by order (see github.com/ethereum/solidity/pull/732)
Expression
  = Expression ('++' | '--')
```

(continues on next page)

(continued from previous page)

```
  | NewExpression
  | IndexAccess
  | IndexRangeAccess
  | MemberAccess
  | FunctionCall
  | '(' Expression ')'
  | ('!' | '~' | 'delete' | '++' | '--' | '+' | '-') Expression
  | Expression '**' Expression
  | Expression ('*' | '/' | '%') Expression
  | Expression ('+' | '-') Expression
  | Expression ('<<' | '>>') Expression
  | Expression '&' Expression
  | Expression '^' Expression
  | Expression '|' Expression
  | Expression ('<' | '>' | '<=' | '>=') Expression
  | Expression ('==' | '!=') Expression
  | Expression '&&' Expression
  | Expression '||' Expression
  | Expression '?' Expression ':' Expression
  | Expression ('=' | '|=' | '^=' | '&=' | '<<=' | '>>=' | '+=' | '-=' | '*=' | '/='␣
↪| '%=') Expression
  | PrimaryExpression

PrimaryExpression = BooleanLiteral
                  | NumberLiteral
                  | HexLiteral
                  | StringLiteral
                  | TupleExpression
                  | Identifier
                  | ElementaryTypeNameExpression

ExpressionList = Expression ( ',' Expression )*
NameValueList = Identifier ':' Expression ( ',' Identifier ':' Expression )*

FunctionCall = Expression '(' FunctionCallArguments ')'
FunctionCallArguments = '{' NameValueList? '}'
                      | ExpressionList?

NewExpression = 'new' TypeName
MemberAccess = Expression '.' Identifier
IndexAccess = Expression '[' Expression? ']'
IndexRangeAccess = Expression '[' Expression? ':' Expression? ']'

BooleanLiteral = 'true' | 'false'
NumberLiteral = ( HexNumber | DecimalNumber ) (' ' NumberUnit)?
NumberUnit = 'wei' | 'szabo' | 'finney' | 'ether'
           | 'seconds' | 'minutes' | 'hours' | 'days' | 'weeks' | 'years'
HexLiteral = 'hex' ('"' ([0-9a-fA-F]{2})* '"' | '\'' ([0-9a-fA-F]{2})* '\'')
StringLiteral = '"' ([^"\r\n\\] | '\\' .)* '"'
Identifier = [a-zA-Z_$] [a-zA-Z_$0-9]*

HexNumber = '0x' [0-9a-fA-F]+
DecimalNumber = [0-9]+ ( '.' [0-9]* )? ( [eE] [0-9]+ )?

TupleExpression = '(' ( Expression? ( ',' Expression? )*  )? ')'
                | '[' ( Expression  ( ',' Expression  )*  )? ']'
```

(continues on next page)

(continued from previous page)

```
ElementaryTypeNameExpression = ElementaryTypeName

ElementaryTypeName = 'address' | 'bool' | 'string' | Int | Uint | Byte | Fixed |␣
↪Ufixed

Int = 'int' | 'int8' | 'int16' | 'int24' | 'int32' | 'int40' | 'int48' | 'int56' |
↪'int64' | 'int72' | 'int80' | 'int88' | 'int96' | 'int104' | 'int112' | 'int120' |
↪'int128' | 'int136' | 'int144' | 'int152' | 'int160' | 'int168' | 'int176' | 'int184
↪' | 'int192' | 'int200' | 'int208' | 'int216' | 'int224' | 'int232' | 'int240' |
↪'int248' | 'int256'

Uint = 'uint' | 'uint8' | 'uint16' | 'uint24' | 'uint32' | 'uint40' | 'uint48' |
↪'uint56' | 'uint64' | 'uint72' | 'uint80' | 'uint88' | 'uint96' | 'uint104' |
↪'uint112' | 'uint120' | 'uint128' | 'uint136' | 'uint144' | 'uint152' | 'uint160' |
↪'uint168' | 'uint176' | 'uint184' | 'uint192' | 'uint200' | 'uint208' | 'uint216' |
↪'uint224' | 'uint232' | 'uint240' | 'uint248' | 'uint256'

Byte = 'byte' | 'bytes' | 'bytes1' | 'bytes2' | 'bytes3' | 'bytes4' | 'bytes5' |
↪'bytes6' | 'bytes7' | 'bytes8' | 'bytes9' | 'bytes10' | 'bytes11' | 'bytes12' |
↪'bytes13' | 'bytes14' | 'bytes15' | 'bytes16' | 'bytes17' | 'bytes18' | 'bytes19' |
↪'bytes20' | 'bytes21' | 'bytes22' | 'bytes23' | 'bytes24' | 'bytes25' | 'bytes26' |
↪'bytes27' | 'bytes28' | 'bytes29' | 'bytes30' | 'bytes31' | 'bytes32'

Fixed = 'fixed' | ( 'fixed' [0-9]+ 'x' [0-9]+ )

Ufixed = 'ufixed' | ( 'ufixed' [0-9]+ 'x' [0-9]+ )


AssemblyBlock = '{' AssemblyStatement* '}'

AssemblyStatement = AssemblyBlock
                  | AssemblyFunctionDefinition
                  | AssemblyVariableDeclaration
                  | AssemblyAssignment
                  | AssemblyIf
                  | AssemblyExpression
                  | AssemblySwitch
                  | AssemblyForLoop
                  | AssemblyBreakContinue
                  | AssemblyLeave
AssemblyFunctionDefinition =
    'function' Identifier '(' AssemblyIdentifierList? ')'
    ( '->' AssemblyIdentifierList )? AssemblyBlock
AssemblyVariableDeclaration = 'let' AssemblyIdentifierList ( ':=' AssemblyExpression␣
↪)?
AssemblyAssignment = AssemblyIdentifierList ':=' AssemblyExpression
AssemblyExpression = AssemblyFunctionCall | Identifier | Literal
AssemblyIf = 'if' AssemblyExpression AssemblyBlock
AssemblySwitch = 'switch' AssemblyExpression ( AssemblyCase+ AssemblyDefault? |␣
↪AssemblyDefault )
AssemblyCase = 'case' Literal AssemblyBlock
AssemblyDefault = 'default' AssemblyBlock
AssemblyForLoop = 'for' AssemblyBlock AssemblyExpression AssemblyBlock AssemblyBlock
AssemblyBreakContinue = 'break' | 'continue'
AssemblyLeave = 'leave'
AssemblyFunctionCall = Identifier '(' ( AssemblyExpression ( ',' AssemblyExpression␣
↪)* )? ')'
```

(continues on next page)

(continued from previous page)

```
AssemblyIdentifierList = Identifier ( ',' Identifier )*
```

2.4.9 Solidity v0.5.0 Breaking Changes

This section highlights the main breaking changes introduced in Solidity version 0.5.0, along with the reasoning behind the changes and how to update affected code. For the full list check the release changelog.

Note: Contracts compiled with Solidity v0.5.0 can still interface with contracts and even libraries compiled with older versions without recompiling or redeploying them. Changing the interfaces to include data locations and visibility and mutability specifiers suffices. See the *Interoperability With Older Contracts* section below.

Semantic Only Changes

This section lists the changes that are semantic-only, thus potentially hiding new and different behavior in existing code.

- Signed right shift now uses proper arithmetic shift, i.e. rounding towards negative infinity, instead of rounding towards zero. Signed and unsigned shift will have dedicated opcodes in Constantinople, and are emulated by Solidity for the moment.
- The `continue` statement in a `do...while` loop now jumps to the condition, which is the common behavior in such cases. It used to jump to the loop body. Thus, if the condition is false, the loop terminates.
- The functions `.call()`, `.delegatecall()` and `.staticcall()` do not pad anymore when given a single `bytes` parameter.
- Pure and view functions are now called using the opcode `STATICCALL` instead of `CALL` if the EVM version is Byzantium or later. This disallows state changes on the EVM level.
- The ABI encoder now properly pads byte arrays and strings from calldata (`msg.data` and external function parameters) when used in external function calls and in `abi.encode`. For unpadded encoding, use `abi.encodePacked`.
- The ABI decoder reverts in the beginning of functions and in `abi.decode()` if passed calldata is too short or points out of bounds. Note that dirty higher order bits are still simply ignored.
- Forward all available gas with external function calls starting from Tangerine Whistle.

Semantic and Syntactic Changes

This section highlights changes that affect syntax and semantics.

- The functions `.call()`, `.delegatecall()`, `staticcall()`, `keccak256()`, `sha256()` and `ripemd160()` now accept only a single `bytes` argument. Moreover, the argument is not padded. This was changed to make more explicit and clear how the arguments are concatenated. Change every `.call()` (and family) to a `.call("")` and every `.call(signature, a, b, c)` to use `.call(abi.encodeWithSignature(signature, a, b, c))` (the last one only works for value types). Change every `keccak256(a, b, c)` to `keccak256(abi.encodePacked(a, b, c))`. Even though it is not a breaking change, it is suggested that developers change `x.call(bytes4(keccak256("f(uint256)"), a, b)` to `x.call(abi.encodeWithSignature("f(uint256)", a, b))`.

- Functions `.call()`, `.delegatecall()` and `.staticcall()` now return `(bool, bytes memory)` to provide access to the return data. Change `bool success = otherContract.call("f")` to `(bool success, bytes memory data) = otherContract.call("f")`.
- Solidity now implements C99-style scoping rules for function local variables, that is, variables can only be used after they have been declared and only in the same or nested scopes. Variables declared in the initialization block of a `for` loop are valid at any point inside the loop.

Explicitness Requirements

This section lists changes where the code now needs to be more explicit. For most of the topics the compiler will provide suggestions.

- Explicit function visibility is now mandatory. Add `public` to every function and constructor, and `external` to every fallback or interface function that does not specify its visibility already.
- Explicit data location for all variables of struct, array or mapping types is now mandatory. This is also applied to function parameters and return variables. For example, change `uint[] x = m_x` to `uint[] storage x = m_x`, and `function f(uint[][] x)` to `function f(uint[][] memory x)` where `memory` is the data location and might be replaced by `storage` or `calldata` accordingly. Note that `external` functions require parameters with a data location of `calldata`.
- Contract types do not include `address` members anymore in order to separate the namespaces. Therefore, it is now necessary to explicitly convert values of contract type to addresses before using an `address` member. Example: if `c` is a contract, change `c.transfer(...)` to `address(c).transfer(...)`, and `c.balance` to `address(c).balance`.
- Explicit conversions between unrelated contract types are now disallowed. You can only convert from a contract type to one of its base or ancestor types. If you are sure that a contract is compatible with the contract type you want to convert to, although it does not inherit from it, you can work around this by converting to `address` first. Example: if `A` and `B` are contract types, `B` does not inherit from `A` and `b` is a contract of type `B`, you can still convert `b` to type `A` using `A(address(b))`. Note that you still need to watch out for matching payable fallback functions, as explained below.
- The `address` type was split into `address` and `address payable`, where only `address payable` provides the `transfer` function. An `address payable` can be directly converted to an `address`, but the other way around is not allowed. Converting `address` to `address payable` is possible via conversion through `uint160`. If `c` is a contract, `address(c)` results in `address payable` only if `c` has a payable fallback function. If you use the *withdraw pattern*, you most likely do not have to change your code because `transfer` is only used on `msg.sender` instead of stored addresses and `msg.sender` is an `address payable`.
- Conversions between `bytesX` and `uintY` of different size are now disallowed due to `bytesX` padding on the right and `uintY` padding on the left which may cause unexpected conversion results. The size must now be adjusted within the type before the conversion. For example, you can convert a `bytes4` (4 bytes) to a `uint64` (8 bytes) by first converting the `bytes4` variable to `bytes8` and then to `uint64`. You get the opposite padding when converting through `uint32`.
- Using `msg.value` in non-payable functions (or introducing it via a modifier) is disallowed as a security feature. Turn the function into `payable` or create a new internal function for the program logic that uses `msg.value`.
- For clarity reasons, the command line interface now requires `-` if the standard input is used as source.

Deprecated Elements

This section lists changes that deprecate prior features or syntax. Note that many of these changes were already enabled in the experimental mode `v0.5.0`.

Command Line and JSON Interfaces

- The command line option `--formal` (used to generate Why3 output for further formal verification) was deprecated and is now removed. A new formal verification module, the SMTChecker, is enabled via `pragma experimental SMTChecker;`.
- The command line option `--julia` was renamed to `--yul` due to the renaming of the intermediate language `Julia` to `Yul`.
- The `--clone-bin` and `--combined-json clone-bin` command line options were removed.
- Remappings with empty prefix are disallowed.
- The JSON AST fields `constant` and `payable` were removed. The information is now present in the `stateMutability` field.
- The JSON AST field `isConstructor` of the `FunctionDefinition` node was replaced by a field called `kind` which can have the value `"constructor"`, `"fallback"` or `"function"`.
- In unlinked binary hex files, library address placeholders are now the first 36 hex characters of the keccak256 hash of the fully qualified library name, surrounded by `$...$`. Previously, just the fully qualified library name was used. This reduces the chances of collisions, especially when long paths are used. Binary files now also contain a list of mappings from these placeholders to the fully qualified names.

Constructors

- Constructors must now be defined using the `constructor` keyword.
- Calling base constructors without parentheses is now disallowed.
- Specifying base constructor arguments multiple times in the same inheritance hierarchy is now disallowed.
- Calling a constructor with arguments but with wrong argument count is now disallowed. If you only want to specify an inheritance relation without giving arguments, do not provide parentheses at all.

Functions

- Function `callcode` is now disallowed (in favor of `delegatecall`). It is still possible to use it via inline assembly.
- `suicide` is now disallowed (in favor of `selfdestruct`).
- `sha3` is now disallowed (in favor of `keccak256`).
- `throw` is now disallowed (in favor of `revert`, `require` and `assert`).

Conversions

- Explicit and implicit conversions from decimal literals to `bytesXX` types is now disallowed.
- Explicit and implicit conversions from hex literals to `bytesXX` types of different size is now disallowed.

Literals and Suffixes

- The unit denomination `years` is now disallowed due to complications and confusions about leap years.
- Trailing dots that are not followed by a number are now disallowed.

- Combining hex numbers with unit denominations (e.g. `0x1e wei`) is now disallowed.
- The prefix `0X` for hex numbers is disallowed, only `0x` is possible.

Variables

- Declaring empty structs is now disallowed for clarity.
- The `var` keyword is now disallowed to favor explicitness.
- Assignments between tuples with different number of components is now disallowed.
- Values for constants that are not compile-time constants are disallowed.
- Multi-variable declarations with mismatching number of values are now disallowed.
- Uninitialized storage variables are now disallowed.
- Empty tuple components are now disallowed.
- Detecting cyclic dependencies in variables and structs is limited in recursion to 256.
- Fixed-size arrays with a length of zero are now disallowed.

Syntax

- Using `constant` as function state mutability modifier is now disallowed.
- Boolean expressions cannot use arithmetic operations.
- The unary + operator is now disallowed.
- Literals cannot anymore be used with `abi.encodePacked` without prior conversion to an explicit type.
- Empty return statements for functions with one or more return values are now disallowed.
- The "loose assembly" syntax is now disallowed entirely, that is, jump labels, jumps and non-functional instructions cannot be used anymore. Use the new `while`, `switch` and `if` constructs instead.
- Functions without implementation cannot use modifiers anymore.
- Function types with named return values are now disallowed.
- Single statement variable declarations inside if/while/for bodies that are not blocks are now disallowed.
- New keywords: `calldata` and `constructor`.
- New reserved keywords: `alias`, `apply`, `auto`, `copyof`, `define`, `immutable`, `implements`, `macro`, `mutable`, `override`, `partial`, `promise`, `reference`, `sealed`, `sizeof`, `supports`, `typedef` and `unchecked`.

Interoperability With Older Contracts

It is still possible to interface with contracts written for Solidity versions prior to v0.5.0 (or the other way around) by defining interfaces for them. Consider you have the following pre-0.5.0 contract already deployed:

```
// This will not compile with the current version of the compiler
pragma solidity ^0.4.25;
contract OldContract {
    function someOldFunction(uint8 a) {
```

(continues on next page)

(continued from previous page)

```
        //...
    }
    function anotherOldFunction() constant returns (bool) {
        //...
    }
    // ...
}
```

This will no longer compile with Solidity v0.5.0. However, you can define a compatible interface for it:

```
pragma solidity >=0.5.0 <0.7.0;
interface OldContract {
    function someOldFunction(uint8 a) external;
    function anotherOldFunction() external returns (bool);
}
```

Note that we did not declare `anotherOldFunction` to be `view`, despite it being declared `constant` in the original contract. This is due to the fact that starting with Solidity v0.5.0 `staticcall` is used to call `view` functions. Prior to v0.5.0 the `constant` keyword was not enforced, so calling a function declared `constant` with `staticcall` may still revert, since the `constant` function may still attempt to modify storage. Consequently, when defining an interface for older contracts, you should only use `view` in place of `constant` in case you are absolutely sure that the function will work with `staticcall`.

Given the interface defined above, you can now easily use the already deployed pre-0.5.0 contract:

```
pragma solidity >=0.5.0 <0.7.0;

interface OldContract {
    function someOldFunction(uint8 a) external;
    function anotherOldFunction() external returns (bool);
}

contract NewContract {
    function doSomething(OldContract a) public returns (bool) {
        a.someOldFunction(0x42);
        return a.anotherOldFunction();
    }
}
```

Similarly, pre-0.5.0 libraries can be used by defining the functions of the library without implementation and supplying the address of the pre-0.5.0 library during linking (see *Using the Commandline Compiler* for how to use the commandline compiler for linking):

```
// This will not compile after 0.6.0
pragma solidity >=0.5.0 <0.5.99;

library OldLibrary {
    function someFunction(uint8 a) public returns(bool);
}

contract NewContract {
    function f(uint8 a) public returns (bool) {
        return OldLibrary.someFunction(a);
    }
}
```

Example

The following example shows a contract and its updated version for Solidity v0.5.0 with some of the changes listed in this section.

Old version:

```
// This will not compile
pragma solidity ^0.4.25;

contract OtherContract {
    uint x;
    function f(uint y) external {
        x = y;
    }
    function() payable external {}
}

contract Old {
    OtherContract other;
    uint myNumber;

    // Function mutability not provided, not an error.
    function someInteger() internal returns (uint) { return 2; }

    // Function visibility not provided, not an error.
    // Function mutability not provided, not an error.
    function f(uint x) returns (bytes) {
        // Var is fine in this version.
        var z = someInteger();
        x += z;
        // Throw is fine in this version.
        if (x > 100)
            throw;
        bytes b = new bytes(x);
        y = -3 >> 1;
        // y == -1 (wrong, should be -2)
        do {
            x += 1;
            if (x > 10) continue;
            // 'Continue' causes an infinite loop.
        } while (x < 11);
        // Call returns only a Bool.
        bool success = address(other).call("f");
        if (!success)
            revert();
        else {
            // Local variables could be declared after their use.
            int y;
        }
        return b;
    }

    // No need for an explicit data location for 'arr'
    function g(uint[] arr, bytes8 x, OtherContract otherContract) public {
        otherContract.transfer(1 ether);

        // Since uint32 (4 bytes) is smaller than bytes8 (8 bytes),
```

(continues on next page)

(continued from previous page)

```
        // the first 4 bytes of x will be lost. This might lead to
        // unexpected behavior since bytesX are right padded.
        uint32 y = uint32(x);
        myNumber += y + msg.value;
    }
}
```

New version:

```
pragma solidity >=0.5.0 <0.7.0;

contract OtherContract {
    uint x;
    function f(uint y) external {
        x = y;
    }
    receive() payable external {}
}

contract New {
    OtherContract other;
    uint myNumber;

    // Function mutability must be specified.
    function someInteger() internal pure returns (uint) { return 2; }

    // Function visibility must be specified.
    // Function mutability must be specified.
    function f(uint x) public returns (bytes memory) {
        // The type must now be explicitly given.
        uint z = someInteger();
        x += z;
        // Throw is now disallowed.
        require(x > 100);
        int y = -3 >> 1;
        require(y == -2);
        do {
            x += 1;
            if (x > 10) continue;
            // 'Continue' jumps to the condition below.
        } while (x < 11);

        // Call returns (bool, bytes).
        // Data location must be specified.
        (bool success, bytes memory data) = address(other).call("f");
        if (!success)
            revert();
        return data;
    }

    using address_make_payable for address;
    // Data location for 'arr' must be specified
    function g(uint[] memory /* arr */, bytes8 x, OtherContract otherContract,␣
↪address unknownContract) public payable {
        // 'otherContract.transfer' is not provided.
        // Since the code of 'OtherContract' is known and has the fallback
        // function, address(otherContract) has type 'address payable'.
```

(continues on next page)

(continued from previous page)

```
        address(otherContract).transfer(1 ether);

        // 'unknownContract.transfer' is not provided.
        // 'address(unknownContract).transfer' is not provided
        // since 'address(unknownContract)' is not 'address payable'.
        // If the function takes an 'address' which you want to send
        // funds to, you can convert it to 'address payable' via 'uint160'.
        // Note: This is not recommended and the explicit type
        // 'address payable' should be used whenever possible.
        // To increase clarity, we suggest the use of a library for
        // the conversion (provided after the contract in this example).
        address payable addr = unknownContract.make_payable();
        require(addr.send(1 ether));

        // Since uint32 (4 bytes) is smaller than bytes8 (8 bytes),
        // the conversion is not allowed.
        // We need to convert to a common size first:
        bytes4 x4 = bytes4(x); // Padding happens on the right
        uint32 y = uint32(x4); // Conversion is consistent
        // 'msg.value' cannot be used in a 'non-payable' function.
        // We need to make the function payable
        myNumber += y + msg.value;
    }
}

// We can define a library for explicitly converting ``address``
// to ``address payable`` as a workaround.
library address_make_payable {
    function make_payable(address x) internal pure returns (address payable) {
        return address(uint160(x));
    }
}
```

2.5 NatSpec Format

Solidity contracts can use a special form of comments to provide rich documentation for functions, return variables and more. This special form is named the Ethereum Natural Language Specification Format (NatSpec).

This documentation is segmented into developer-focused messages and end-user-facing messages. These messages may be shown to the end user (the human) at the time that they will interact with the contract (i.e. sign a transaction).

It is recommended that Solidity contracts are fully annotated using NatSpec for all public interfaces (everything in the ABI).

NatSpec includes the formatting for comments that the smart contract author will use, and which are understood by the Solidity compiler. Also detailed below is output of the Solidity compiler, which extracts these comments into a machine-readable format.

2.5.1 Documentation Example

Documentation is inserted above each `class`, `interface` and `function` using the doxygen notation format.

- For Solidity you may choose `///` for single or multi-line comments, or `/**` and ending with `*/`.
- For Vyper, use `"""` indented to the inner contents with bare comments. See Vyper documentation.

The following example shows a contract and a function using all available tags.

Note: NatSpec currently does NOT apply to public state variables (see solidity#3418), even if they are declared public and therefore do affect the ABI.

The Solidity compiler only interprets tags if they are external or public. You are welcome to use similar comments for your internal and private functions, but those will not be parsed.

```
pragma solidity >=0.5.0 <0.7.0;

/// @title A simulator for trees
/// @author Larry A. Gardner
/// @notice You can use this contract for only the most basic simulation
/// @dev All function calls are currently implemented without side effects
contract Tree {
    /// @author Mary A. Botanist
    /// @notice Calculate tree age in years, rounded up, for live trees
    /// @dev The Alexandr N. Tetearing algorithm could increase precision
    /// @param rings The number of rings from dendrochronological sample
    /// @return age in years, rounded up for partial years
    function age(uint256 rings) external pure returns (uint256) {
        return rings + 1;
    }
}
```

2.5.2 Tags

All tags are optional. The following table explains the purpose of each NatSpec tag and where it may be used. As a special case, if no tags are used then the Solidity compiler will interpret a */// or /*** comment in the same way as if it were tagged with *@notice*.

Tag		Context
`@title`	A title that should describe the contract/interface	contract, interface
`@author`	The name of the author	contract, interface, function
`@notice`	Explain to an end user what this does	contract, interface, function
`@dev`	Explain to a developer any extra details	contract, interface, function
`@param`	Documents a parameter just like in doxygen (must be followed by parameter name)	function
`@return`	Documents the return variables of a contract's function	function

If your function returns multiple values, like `(int quotient, int remainder)` then use multiple `@return` statements in the same format as the `@param` statements.

Dynamic expressions

The Solidity compiler will pass through NatSpec documentation from your Solidity source code to the JSON output as described in this guide. The consumer of this JSON output, for example the end-user client software, may present this to the end-user directly or it may apply some pre-processing.

For example, some client software will render:

```
/// @notice This function will multiply `a` by 7
```

to the end-user as:

```
This function will multiply 10 by 7
```

if a function is being called and the input `a` is assigned a value of 10.

Specifying these dynamic expressions is outside the scope of the Solidity documentation and you may read more at the radspec project.

Inheritance Notes

Currently it is undefined whether a contract with a function having no NatSpec will inherit the NatSpec of a parent contract/interface for that same function.

2.5.3 Documentation Output

When parsed by the compiler, documentation such as the one from the above example will produce two different JSON files. One is meant to be consumed by the end user as a notice when a function is executed and the other to be used by the developer.

If the above contract is saved as `ex1.sol` then you can generate the documentation using:

```
solc --userdoc --devdoc ex1.sol
```

And the output is below.

User Documentation

The above documentation will produce the following user documentation JSON file as output:

```
{
  "methods" :
  {
    "age(uint256)" :
    {
      "notice" : "Calculate tree age in years, rounded up, for live trees"
    }
  },
  "notice" : "You can use this contract for only the most basic simulation"
}
```

Note that the key by which to find the methods is the function's canonical signature as defined in the Contract ABI and not simply the function's name.

Developer Documentation

Apart from the user documentation file, a developer documentation JSON file should also be produced and should look like this:

```
{
  "author" : "Larry A. Gardner",
  "details" : "All function calls are currently implemented without side effects",
  "methods" :
  {
    "age(uint256)" :
    {
      "author" : "Mary A. Botanist",
      "details" : "The Alexandr N. Tetearing algorithm could increase precision",
      "params" :
      {
        "rings" : "The number of rings from dendrochronological sample"
      },
      "return" : "age in years, rounded up for partial years"
    }
  },
  "title" : "A simulator for trees"
}
```

2.6 Resources

2.6.1 General

- Ethereum
- Changelog
- Source Code
- Ethereum Stackexchange
- Language Users Chat
- Compiler Developers Chat

2.6.2 Solidity Integrations

- Generic:
  - **EthFiddle** Solidity IDE in the Browser. Write and share your Solidity code. Uses server-side components.
  - **Remix** Browser-based IDE with integrated compiler and Solidity runtime environment without server-side components.
  - **Solhint** Solidity linter that provides security, style guide and best practice rules for smart contract validation.
  - **Solidity IDE** Browser-based IDE with integrated compiler, Ganache and local file system support.
  - **Solium** Linter to identify and fix style and security issues in Solidity.
  - **Superblocks Lab** Browser-based IDE. Built-in browser-based VM and Metamask integration (one click deployment to Testnet/Mainnet).
- Atom:
  - **Etheratom** Plugin for the Atom editor that features syntax highlighting, compilation and a runtime environment (Backend node & VM compatible).

  - **Atom Solidity Linter** Plugin for the Atom editor that provides Solidity linting.
  - **Atom Solium Linter** Configurable Solidty linter for Atom using Solium as a base.

- Eclipse:
  - **YAKINDU Solidity Tools** Eclipse based IDE. Features context sensitive code completion and help, code navigation, syntax coloring, built in compiler, quick fixes and templates.
- Emacs:
  - **Emacs Solidity** Plugin for the Emacs editor providing syntax highlighting and compilation error reporting.
- IntelliJ:
  - **IntelliJ IDEA plugin** Solidity plugin for IntelliJ IDEA (and all other JetBrains IDEs)
- Sublime:
  - **Package for SublimeText - Solidity language syntax** Solidity syntax highlighting for SublimeText editor.
- Vim:
  - **Vim Solidity** Plugin for the Vim editor providing syntax highlighting.
  - **Vim Syntastic** Plugin for the Vim editor providing compile checking.
- Visual Studio Code:
  - **Visual Studio Code extension** Solidity plugin for Microsoft Visual Studio Code that includes syntax highlighting and the Solidity compiler.

Discontinued:

- **Mix IDE** Qt based IDE for designing, debugging and testing solidity smart contracts.
- **Ethereum Studio** Specialized web IDE that also provides shell access to a complete Ethereum environment.
- **Visual Studio Extension** Solidity plugin for Microsoft Visual Studio that includes the Solidity compiler.

2.6.3 Solidity Tools

- **Dapp** Build tool, package manager, and deployment assistant for Solidity.
- **Solidity REPL** Try Solidity instantly with a command-line Solidity console.
- **solgraph** Visualize Solidity control flow and highlight potential security vulnerabilities.
- **Doxity** Documentation Generator for Solidity.
- **evmdis** EVM Disassembler that performs static analysis on the bytecode to provide a higher level of abstraction than raw EVM operations.
- **ABI to solidity interface converter** A script for generating contract interfaces from the ABI of a smart contract.
- **Securify** Fully automated online static analyzer for smart contracts, providing a security report based on vulnerability patterns.
- **Sūrya** Utility tool for smart contract systems, offering a number of visual outputs and information about the contracts' structure. Also supports querying the function call graph.
- **EVM Lab** Rich tool package to interact with the EVM. Includes a VM, Etherchain API, and a trace-viewer with gas cost display.

- **Universal Mutator** A tool for mutation generation, with configurable rules and support for Solidity and Vyper.
- **PIET** A tool to develop, audit and use solidity smart contracts through a simple graphical interface.

Note: Information like variable names, comments, and source code formatting is lost in the compilation process and it is not possible to completely recover the original source code. Decompiling smart contracts to view the original source code might not be possible, or the end result that useful.

2.6.4 Third-Party Solidity Parsers and Grammars

- **solidity-parser** Solidity parser for JavaScript
- **Solidity Grammar for ANTLR 4** Solidity grammar for the ANTLR 4 parser generator

2.7 Using the compiler

2.7.1 Using the Commandline Compiler

Note: This section does not apply to solcjs, not even if it is used in commandline mode.

One of the build targets of the Solidity repository is `solc`, the solidity commandline compiler. Using `solc --help` provides you with an explanation of all options. The compiler can produce various outputs, ranging from simple binaries and assembly over an abstract syntax tree (parse tree) to estimations of gas usage. If you only want to compile a single file, you run it as `solc --bin sourceFile.sol` and it will print the binary. If you want to get some of the more advanced output variants of `solc`, it is probably better to tell it to output everything to separate files using `solc -o outputDirectory --bin --ast-json --asm sourceFile.sol`.

Before you deploy your contract, activate the optimizer when compiling using `solc --optimize --bin sourceFile.sol`. By default, the optimizer will optimize the contract assuming it is called 200 times across its lifetime (more specifically, it assumes each opcode is executed around 200 times). If you want the initial contract deployment to be cheaper and the later function executions to be more expensive, set it to `--optimize-runs=1`. If you expect many transactions and do not care for higher deployment cost and output size, set `--optimize-runs` to a high number. This parameter has effects on the following (this might change in the future):

- the size of the binary search in the function dispatch routine
- the way constants like large numbers or strings are stored

The commandline compiler will automatically read imported files from the filesystem, but it is also possible to provide path redirects using `prefix=path` in the following way:

```
solc github.com/ethereum/dapp-bin/=/usr/local/lib/dapp-bin/ file.sol
```

This essentially instructs the compiler to search for anything starting with `github.com/ethereum/dapp-bin/` under `/usr/local/lib/dapp-bin`. `solc` will not read files from the filesystem that lie outside of the remapping targets and outside of the directories where explicitly specified source files reside, so things like `import "/etc/passwd";` only work if you add `/=/` as a remapping.

An empty remapping prefix is not allowed.

If there are multiple matches due to remappings, the one with the longest common prefix is selected.

For security reasons the compiler has restrictions what directories it can access. Paths (and their subdirectories) of source files specified on the commandline and paths defined by remappings are allowed for import statements, but everything else is rejected. Additional paths (and their subdirectories) can be allowed via the `--allow-paths /sample/path,/another/sample/path` switch.

If your contracts use *libraries*, you will notice that the bytecode contains substrings of the form `__$53aea86b7d70b31448b230b20ae141a537$__`. These are placeholders for the actual library addresses. The placeholder is a 34 character prefix of the hex encoding of the keccak256 hash of the fully qualified library name. The bytecode file will also contain lines of the form `// <placeholder> -> <fq library name>` at the end to help identify which libraries the placeholders represent. Note that the fully qualified library name is the path of its source file and the library name separated by `:`. You can use `solc` as a linker meaning that it will insert the library addresses for you at those points:

Either add `--libraries "file.sol:Math:0x1234567890123456789012345678901234567890 file.sol:Heap:0xabCD567890123456789012345678901234567890"` to your command to provide an address for each library or store the string in a file (one library per line) and run `solc` using `--libraries fileName`.

If `solc` is called with the option `--link`, all input files are interpreted to be unlinked binaries (hex-encoded) in the `__$53aea86b7d70b31448b230b20ae141a537$__`-format given above and are linked in-place (if the input is read from stdin, it is written to stdout). All options except `--libraries` are ignored (including `-o`) in this case.

If `solc` is called with the option `--standard-json`, it will expect a JSON input (as explained below) on the standard input, and return a JSON output on the standard output. This is the recommended interface for more complex and especially automated uses. The process will always terminate in a "success" state and report any errors via the JSON output.

Note: The library placeholder used to be the fully qualified name of the library itself instead of the hash of it. This format is still supported by `solc --link` but the compiler will no longer output it. This change was made to reduce the likelihood of a collision between libraries, since only the first 36 characters of the fully qualified library name could be used.

2.7.2 Setting the EVM version to target

When you compile your contract code you can specify the Ethereum virtual machine version to compile for to avoid particular features or behaviours.

Warning: Compiling for the wrong EVM version can result in wrong, strange and failing behaviour. Please ensure, especially if running a private chain, that you use matching EVM versions.

On the command line, you can select the EVM version as follows:

```
solc --evm-version <VERSION> contract.sol
```

In the *standard JSON interface*, use the `"evmVersion"` key in the `"settings"` field:

```
{
  "sources": { ... },
  "settings": {
    "optimizer": { ... },
    "evmVersion": "<VERSION>"
  }
}
```

Target options

Below is a list of target EVM versions and the compiler-relevant changes introduced at each version. Backward compatibility is not guaranteed between each version.

- `homestead` (oldest version)
- **`tangerineWhistle`**
  - Gas cost for access to other accounts increased, relevant for gas estimation and the optimizer.
  - All gas sent by default for external calls, previously a certain amount had to be retained.
- **`spuriousDragon`**
  - Gas cost for the `exp` opcode increased, relevant for gas estimation and the optimizer.
- **`byzantium`**
  - Opcodes `returndatacopy`, `returndatasize` and `staticcall` are available in assembly.
  - The `staticcall` opcode is used when calling non-library view or pure functions, which prevents the functions from modifying state at the EVM level, i.e., even applies when you use invalid type conversions.
  - It is possible to access dynamic data returned from function calls.
  - `revert` opcode introduced, which means that `revert()` will not waste gas.
- **`constantinople`**
  - Opcodes ``create2`, ``extcodehash``, `shl`, `shr` and `sar` are available in assembly.
  - Shifting operators use shifting opcodes and thus need less gas.
- **`petersburg`**
  - The compiler behaves the same way as with constantinople.
- **`istanbul` (default)**
  - Opcodes `chainid` and `selfbalance` are available in assembly.
- `berlin` (**experimental**)

2.7.3 Compiler Input and Output JSON Description

The recommended way to interface with the Solidity compiler especially for more complex and automated setups is the so-called JSON-input-output interface. The same interface is provided by all distributions of the compiler.

The fields are generally subject to change, some are optional (as noted), but we try to only make backwards compatible changes.

The compiler API expects a JSON formatted input and outputs the compilation result in a JSON formatted output. The standard error output is not used and the process will always terminate in a “success” state, even if there were errors. Errors are always reported as part of the JSON output.

The following subsections describe the format through an example. Comments are of course not permitted and used here only for explanatory purposes.

Input Description

```
{
  // Required: Source code language. Currently supported are "Solidity" and "Yul".
  "language": "Solidity",
  // Required
  "sources":
  {
    // The keys here are the "global" names of the source files,
    // imports can use other files via remappings (see below).
    "myFile.sol":
    {
      // Optional: keccak256 hash of the source file
      // It is used to verify the retrieved content if imported via URLs.
      "keccak256": "0x123...",
      // Required (unless "content" is used, see below): URL(s) to the source file.
      // URL(s) should be imported in this order and the result checked against the
      // keccak256 hash (if available). If the hash doesn't match or none of the
      // URL(s) result in success, an error should be raised.
      // Using the commandline interface only filesystem paths are supported.
      // With the JavaScript interface the URL will be passed to the user-supplied
      // read callback, so any URL supported by the callback can be used.
      "urls":
      [
        "bzzr://56ab...",
        "ipfs://Qma...",
        "/tmp/path/to/file.sol"
        // If files are used, their directories should be added to the command line␣
↪via
        // `--allow-paths <path>`.
      ]
    },
    "mortal":
    {
      // Optional: keccak256 hash of the source file
      "keccak256": "0x234...",
      // Required (unless "urls" is used): literal contents of the source file
      "content": "contract mortal is owned { function kill() { if (msg.sender ==␣
↪owner) selfdestruct(owner); } }"
    }
  },
  // Optional
  "settings":
  {
    // Optional: Sorted list of remappings
    "remappings": [ ":g=/dir" ],
    // Optional: Optimizer settings
    "optimizer": {
      // disabled by default
      "enabled": true,
      // Optimize for how many times you intend to run the code.
      // Lower values will optimize more for initial deployment cost, higher
      // values will optimize more for high-frequency usage.
      "runs": 200,
      // Switch optimizer components on or off in detail.
      // The "enabled" switch above provides two defaults which can be
      // tweaked here. If "details" is given, "enabled" can be omitted.
```

(continues on next page)

(continued from previous page)

```
    "details": {
      // The peephole optimizer is always on if no details are given,
      // use details to switch it off.
      "peephole": true,
      // The unused jumpdest remover is always on if no details are given,
      // use details to switch it off.
      "jumpdestRemover": true,
      // Sometimes re-orders literals in commutative operations.
      "orderLiterals": false,
      // Removes duplicate code blocks
      "deduplicate": false,
      // Common subexpression elimination, this is the most complicated step but
      // can also provide the largest gain.
      "cse": false,
      // Optimize representation of literal numbers and strings in code.
      "constantOptimizer": false,
      // The new Yul optimizer. Mostly operates on the code of ABIEncoderV2.
      // It can only be activated through the details here.
      "yul": false,
      // Tuning options for the Yul optimizer.
      "yulDetails": {
        // Improve allocation of stack slots for variables, can free up stack slots early.
        // Activated by default if the Yul optimizer is activated.
        "stackAllocation": true
      }
    }
  },
  // Version of the EVM to compile for.
  // Affects type checking and code generation. Can be homestead,
  // tangerineWhistle, spuriousDragon, byzantium, constantinople, petersburg, istanbul or berlin
  "evmVersion": "byzantium",
  // Optional: Debugging settings
  "debug": {
    // How to treat revert (and require) reason strings. Settings are
    // "default", "strip", "debug" and "verboseDebug".
    // "default" does not inject compiler-generated revert strings and keeps user-supplied ones.
    // "strip" removes all revert strings (if possible, i.e. if literals are used) keeping side-effects
    // "debug" injects strings for compiler-generated internal reverts (not yet implemented)
    // "verboseDebug" even appends further information to user-supplied revert strings (not yet implemented)
    "revertStrings": "default"
  }
  // Metadata settings (optional)
  "metadata": {
    // Use only literal content and not URLs (false by default)
    "useLiteralContent": true,
    // Use the given hash method for the metadata hash that is appended to the bytecode.
    // The metadata hash can be removed from the bytecode via option "none".
    // The other options are "ipfs" and "bzzr1".
    // If the option is omitted, "ipfs" is used by default.
    "bytecodeHash": "ipfs"
```

(continues on next page)

(continued from previous page)

```
    },
    // Addresses of the libraries. If not all libraries are given here,
    // it can result in unlinked objects whose output data is different.
    "libraries": {
      // The top level key is the the name of the source file where the library is used.
      // If remappings are used, this source file should match the global path
      // after remappings were applied.
      // If this key is an empty string, that refers to a global level.
      "myFile.sol": {
        "MyLib": "0x123123..."
      }
    }
    // The following can be used to select desired outputs based
    // on file and contract names.
    // If this field is omitted, then the compiler loads and does type checking,
    // but will not generate any outputs apart from errors.
    // The first level key is the file name and the second level key is the contract name.
    // An empty contract name is used for outputs that are not tied to a contract
    // but to the whole source file like the AST.
    // A star as contract name refers to all contracts in the file.
    // Similarly, a star as a file name matches all files.
    // To select all outputs the compiler can possibly generate, use
    // "outputSelection: { "*": { "*": [ "*" ], "": [ "*" ] } }"
    // but note that this might slow down the compilation process needlessly.
    //
    // The available output types are as follows:
    //
    // File level (needs empty string as contract name):
    //   ast - AST of all source files
    //   legacyAST - legacy AST of all source files
    //
    // Contract level (needs the contract name or "*"):
    //   abi - ABI
    //   devdoc - Developer documentation (natspec)
    //   userdoc - User documentation (natspec)
    //   metadata - Metadata
    //   ir - Yul intermediate representation of the code before optimization
    //   irOptimized - Intermediate representation after optimization
    //   storageLayout - Slots, offsets and types of the contract's state variables.
    //   evm.assembly - New assembly format
    //   evm.legacyAssembly - Old-style assembly format in JSON
    //   evm.bytecode.object - Bytecode object
    //   evm.bytecode.opcodes - Opcodes list
    //   evm.bytecode.sourceMap - Source mapping (useful for debugging)
    //   evm.bytecode.linkReferences - Link references (if unlinked object)
    //   evm.deployedBytecode* - Deployed bytecode (has the same options as evm.bytecode)
    //   evm.methodIdentifiers - The list of function hashes
    //   evm.gasEstimates - Function gas estimates
    //   ewasm.wast - eWASM S-expressions format (not supported at the moment)
    //   ewasm.wasm - eWASM binary format (not supported at the moment)
    //
    // Note that using a using `evm`, `evm.bytecode`, `ewasm`, etc. will select every
    // target part of that output. Additionally, `*` can be used as a wildcard to request everything.
```

(continues on next page)

(continued from previous page)

```
    //
    "outputSelection": {
      "*": {
        "*": [
          "metadata", "evm.bytecode" // Enable the metadata and bytecode outputs of␣
↪every single contract.
          , "evm.bytecode.sourceMap" // Enable the source map output of every single␣
↪contract.
        ],
        "": [
          "ast" // Enable the AST output of every single file.
        ]
      },
      // Enable the abi and opcodes output of MyContract defined in file def.
      "def": {
        "MyContract": [ "abi", "evm.bytecode.opcodes" ]
      }
    }
  }
}
```

Output Description

```
{
  // Optional: not present if no errors/warnings were encountered
  "errors": [
    {
      // Optional: Location within the source file.
      "sourceLocation": {
        "file": "sourceFile.sol",
        "start": 0,
        "end": 100
      ],
      // Optional: Further locations (e.g. places of conflicting declarations)
      "secondarySourceLocations": [
        {
          "file": "sourceFile.sol",
          "start": 64,
          "end": 92,
          "message": "Other declaration is here:"
        }
      ],
      // Mandatory: Error type, such as "TypeError", "InternalCompilerError",
↪"Exception", etc.
      // See below for complete list of types.
      "type": "TypeError",
      // Mandatory: Component where the error originated, such as "general", "ewasm",␣
↪etc.
      "component": "general",
      // Mandatory ("error" or "warning")
      "severity": "error",
      // Mandatory
      "message": "Invalid keyword"
      // Optional: the message formatted with source location
      "formattedMessage": "sourceFile.sol:100: Invalid keyword"
```

(continues on next page)

(continued from previous page)

```
    }
  ],
  // This contains the file-level outputs.
  // It can be limited/filtered by the outputSelection settings.
  "sources": {
    "sourceFile.sol": {
      // Identifier of the source (used in source maps)
      "id": 1,
      // The AST object
      "ast": {},
      // The legacy AST object
      "legacyAST": {}
    }
  },
  // This contains the contract-level outputs.
  // It can be limited/filtered by the outputSelection settings.
  "contracts": {
    "sourceFile.sol": {
      // If the language used has no contract names, this field should equal to an empty string.
      "ContractName": {
        // The Ethereum Contract ABI. If empty, it is represented as an empty array.
        // See https://solidity.readthedocs.io/en/develop/abi-spec.html
        "abi": [],
        // See the Metadata Output documentation (serialised JSON string)
        "metadata": "{...}",
        // User documentation (natspec)
        "userdoc": {},
        // Developer documentation (natspec)
        "devdoc": {},
        // Intermediate representation (string)
        "ir": "",
        // See the Storage Layout documentation.
        "storageLayout": {"storage": [...], "types": {...} },
        // EVM-related outputs
        "evm": {
          // Assembly (string)
          "assembly": "",
          // Old-style assembly (object)
          "legacyAssembly": {},
          // Bytecode and related details.
          "bytecode": {
            // The bytecode as a hex string.
            "object": "00fe",
            // Opcodes list (string)
            "opcodes": "",
            // The source mapping as a string. See the source mapping definition.
            "sourceMap": "",
            // If given, this is an unlinked object.
            "linkReferences": {
              "libraryFile.sol": {
                // Byte offsets into the bytecode.
                // Linking replaces the 20 bytes located there.
                "Library1": [
                  { "start": 0, "length": 20 },
                  { "start": 200, "length": 20 }
                ]
```

(continues on next page)

(continued from previous page)

```
              }
            }
          },
          // The same layout as above.
          "deployedBytecode": { },
          // The list of function hashes
          "methodIdentifiers": {
            "delegate(address)": "5c19a95c"
          },
          // Function gas estimates
          "gasEstimates": {
            "creation": {
              "codeDepositCost": "420000",
              "executionCost": "infinite",
              "totalCost": "infinite"
            },
            "external": {
              "delegate(address)": "25000"
            },
            "internal": {
              "heavyLifting()": "infinite"
            }
          }
        },
        // eWASM related outputs
        "ewasm": {
          // S-expressions format
          "wast": "",
          // Binary format (hex string)
          "wasm": ""
        }
      }
    }
  }
}
```

Error types

1. `JSONError`: JSON input doesn’t conform to the required format, e.g. input is not a JSON object, the language is not supported, etc.
2. `IOError`: IO and import processing errors, such as unresolvable URL or hash mismatch in supplied sources.
3. `ParserError`: Source code doesn’t conform to the language rules.
4. `DocstringParsingError`: The NatSpec tags in the comment block cannot be parsed.
5. `SyntaxError`: Syntactical error, such as `continue` is used outside of a `for` loop.
6. `DeclarationError`: Invalid, unresolvable or clashing identifier names. e.g. `Identifier not found`
7. `TypeError`: Error within the type system, such as invalid type conversions, invalid assignments, etc.
8. `UnimplementedFeatureError`: Feature is not supported by the compiler, but is expected to be supported in future versions.
9. `InternalCompilerError`: Internal bug triggered in the compiler - this should be reported as an issue.
10. `Exception`: Unknown failure during compilation - this should be reported as an issue.

11. `CompilerError`: Invalid use of the compiler stack - this should be reported as an issue.
12. `FatalError`: Fatal error not processed correctly - this should be reported as an issue.
13. `Warning`: A warning, which didn't stop the compilation, but should be addressed if possible.

2.8 Contract Metadata

The Solidity compiler automatically generates a JSON file, the contract metadata, that contains information about the compiled contract. You can use this file to query the compiler version, the sources used, the ABI and NatSpec documentation to more safely interact with the contract and verify its source code.

The compiler appends by default the IPFS hash of the metadata file to the end of the bytecode (for details, see below) of each contract, so that you can retrieve the file in an authenticated way without having to resort to a centralized data provider. The other available options are the Swarm hash and not appending the metadata hash to the bytecode. These can be configured via the *Standard JSON Interface*.

You have to publish the metadata file to IPFS, Swarm, or another service so that others can access it. You create the file by using the `solc --metadata` command that generates a file called `ContractName_meta.json`. It contains IPFS and Swarm references to the source code, so you have to upload all source files and the metadata file.

The metadata file has the following format. The example below is presented in a human-readable way. Properly formatted metadata should use quotes correctly, reduce whitespace to a minimum and sort the keys of all objects to arrive at a unique formatting. Comments are not permitted and used here only for explanatory purposes.

```
{
  // Required: The version of the metadata format
  version: "1",
  // Required: Source code language, basically selects a "sub-version"
  // of the specification
  language: "Solidity",
  // Required: Details about the compiler, contents are specific
  // to the language.
  compiler: {
    // Required for Solidity: Version of the compiler
    version: "0.4.6+commit.2dabbdf0.Emscripten.clang",
    // Optional: Hash of the compiler binary which produced this output
    keccak256: "0x123..."
  },
  // Required: Compilation source files/source units, keys are file names
  sources:
  {
    "myFile.sol": {
      // Required: keccak256 hash of the source file
      "keccak256": "0x123...",
      // Required (unless "content" is used, see below): Sorted URL(s)
      // to the source file, protocol is more or less arbitrary, but a
      // Swarm URL is recommended
      "urls": [ "bzzr://56ab..." ]
    },
    "mortal": {
      // Required: keccak256 hash of the source file
      "keccak256": "0x234...",
      // Required (unless "url" is used): literal contents of the source file
      "content": "contract mortal is owned { function kill() { if (msg.sender ==␣
↪owner) selfdestruct(owner); } }"
    }
```

(continues on next page)

(continued from previous page)

```
  },
  // Required: Compiler settings
  settings:
  {
    // Required for Solidity: Sorted list of remappings
    remappings: [ ":g=/dir" ],
    // Optional: Optimizer settings. The fields "enabled" and "runs" are deprecated
    // and are only given for backwards-compatibility.
    optimizer: {
      enabled: true,
      runs: 500,
      details: {
        // peephole defaults to "true"
        peephole: true,
        // jumpdestRemover defaults to "true"
        jumpdestRemover: true,
        orderLiterals: false,
        deduplicate: false,
        cse: false,
        constantOptimizer: false,
        yul: false,
        yulDetails: {}
      }
    },
    metadata: {
      // Reflects the setting used in the input json, defaults to false
      useLiteralContent: true,
      // Reflects the setting used in the input json, defaults to "ipfs"
      bytecodeHash: "ipfs"
    }
    // Required for Solidity: File and name of the contract or library this
    // metadata is created for.
    compilationTarget: {
      "myFile.sol": "MyContract"
    },
    // Required for Solidity: Addresses for libraries used
    libraries: {
      "MyLib": "0x123123..."
    }
  },
  // Required: Generated information about the contract.
  output:
  {
    // Required: ABI definition of the contract
    abi: [ ... ],
    // Required: NatSpec user documentation of the contract
    userdoc: [ ... ],
    // Required: NatSpec developer documentation of the contract
    devdoc: [ ... ],
  }
}
```

Warning: Since the bytecode of the resulting contract contains the metadata hash by default, any change to the metadata might result in a change of the bytecode. This includes changes to a filename or path, and since the metadata includes a hash of all the sources used, a single whitespace change results in different metadata, and different bytecode.

Note: The ABI definition above has no fixed order. It can change with compiler versions. Starting from Solidity version 0.5.12, though, the array maintains a certain order.

2.8.1 Encoding of the Metadata Hash in the Bytecode

Because we might support other ways to retrieve the metadata file in the future, the mapping `{"ipfs": <IPFS hash>, "solc": <compiler version>}` is stored cbor-encoded. Since the mapping might contain more keys (see below) and the beginning of that encoding is not easy to find, its length is added in a two-byte big-endian encoding. The current version of the Solidity compiler usually adds the following to the end of the deployed bytecode:

```
0xa2
0x64 'i' 'p' 'f' 's' 0x58 0x22 <34 bytes IPFS hash>
0x64 's' 'o' 'l' 'c' 0x43 <3 byte version encoding>
0x00 0x32
```

So in order to retrieve the data, the end of the deployed bytecode can be checked to match that pattern and use the IPFS hash to retrieve the file.

Whereas release builds of solc use a 3 byte encoding of the version as shown above (one byte each for major, minor and patch version number), prerelease builds will instead use a complete version string including commit hash and build date.

Note: The CBOR mapping can also contain other keys, so it is better to fully decode the data instead of relying on it starting with `0xa264`. For example, if any experimental features that affect code generation are used, the mapping will also contain `"experimental": true`.

Note: The compiler currently uses the IPFS hash of the metadata by default, but it may also use the bzzr1 hash or some other hash in the future, so do not rely on this sequence to start with `0xa2 0x64 'i' 'p' 'f' 's'`. We might also add additional data to this CBOR structure, so the best option is to use a proper CBOR parser.

2.8.2 Usage for Automatic Interface Generation and NatSpec

The metadata is used in the following way: A component that wants to interact with a contract (e.g. Mist or any wallet) retrieves the code of the contract, from that the IPFS/Swarm hash of a file which is then retrieved. That file is JSON-decoded into a structure like above.

The component can then use the ABI to automatically generate a rudimentary user interface for the contract.

Furthermore, the wallet can use the NatSpec user documentation to display a confirmation message to the user whenever they interact with the contract, together with requesting authorization for the transaction signature.

For additional information, read *Ethereum Natural Language Specification (NatSpec) format*.

2.8.3 Usage for Source Code Verification

In order to verify the compilation, sources can be retrieved from IPFS/Swarm via the link in the metadata file. The compiler of the correct version (which is checked to be part of the "official" compilers) is invoked on that input with the specified settings. The resulting bytecode is compared to the data of the creation transaction or `CREATE` opcode data. This automatically verifies the metadata since its hash is part of the bytecode. Excess data corresponds to the constructor input data, which should be decoded according to the interface and presented to the user.

2.9 Contract ABI Specification

2.9.1 Basic Design

The Contract Application Binary Interface (ABI) is the standard way to interact with contracts in the Ethereum ecosystem, both from outside the blockchain and for contract-to-contract interaction. Data is encoded according to its type, as described in this specification. The encoding is not self describing and thus requires a schema in order to decode.

We assume the interface functions of a contract are strongly typed, known at compilation time and static. We assume that all contracts will have the interface definitions of any contracts they call available at compile-time.

This specification does not address contracts whose interface is dynamic or otherwise known only at run-time.

2.9.2 Function Selector

The first four bytes of the call data for a function call specifies the function to be called. It is the first (left, high-order in big-endian) four bytes of the Keccak-256 (SHA-3) hash of the signature of the function. The signature is defined as the canonical expression of the basic prototype without data location specifier, i.e. the function name with the parenthesised list of parameter types. Parameter types are split by a single comma - no spaces are used.

Note: The return type of a function is not part of this signature. In *Solidity's function overloading* return types are not considered. The reason is to keep function call resolution context-independent. The *JSON description of the ABI* however contains both inputs and outputs.

2.9.3 Argument Encoding

Starting from the fifth byte, the encoded arguments follow. This encoding is also used in other places, e.g. the return values and also event arguments are encoded in the same way, without the four bytes specifying the function.

2.9.4 Types

The following elementary types exist:

- `uint<M>`: unsigned integer type of `M` bits, `0 < M <= 256`, `M % 8 == 0`. e.g. `uint32`, `uint8`, `uint256`.
- `int<M>`: two's complement signed integer type of `M` bits, `0 < M <= 256`, `M % 8 == 0`.
- `address`: equivalent to `uint160`, except for the assumed interpretation and language typing. For computing the function selector, `address` is used.
- `uint`, `int`: synonyms for `uint256`, `int256` respectively. For computing the function selector, `uint256` and `int256` have to be used.

- `bool`: equivalent to `uint8` restricted to the values 0 and 1. For computing the function selector, `bool` is used.
- `fixed<M>x<N>`: signed fixed-point decimal number of `M` bits, `8 <= M <= 256`, `M % 8 ==0`, and `0 < N <= 80`, which denotes the value `v` as `v / (10 ** N)`.
- `ufixed<M>x<N>`: unsigned variant of `fixed<M>x<N>`.
- `fixed`, `ufixed`: synonyms for `fixed128x18`, `ufixed128x18` respectively. For computing the function selector, `fixed128x18` and `ufixed128x18` have to be used.
- `bytes<M>`: binary type of `M` bytes, `0 < M <= 32`.
- `function`: an address (20 bytes) followed by a function selector (4 bytes). Encoded identical to `bytes24`.

The following (fixed-size) array type exists:

- `<type>[M]`: a fixed-length array of `M` elements, `M >= 0`, of the given type.

The following non-fixed-size types exist:

- `bytes`: dynamic sized byte sequence.
- `string`: dynamic sized unicode string assumed to be UTF-8 encoded.
- `<type>[]`: a variable-length array of elements of the given type.

Types can be combined to a tuple by enclosing them inside parentheses, separated by commas:

- `(T1,T2,...,Tn)`: tuple consisting of the types `T1`, …, `Tn`, `n >= 0`

It is possible to form tuples of tuples, arrays of tuples and so on. It is also possible to form zero-tuples (where `n == 0`).

Mapping Solidity to ABI types

Solidity supports all the types presented above with the same names with the exception of tuples. On the other hand, some Solidity types are not supported by the ABI. The following table shows on the left column Solidity types that are not part of the ABI, and on the right column the ABI types that represent them.

Solidity	ABI
address payable	`address`
contract	`address`
enum	smallest `uint` type that is large enough to hold all values For example, an `enum` of 255 values or less is mapped to `uint8` and an `enum` of 256 values is mapped to `uint16`.
struct	`tuple`

2.9.5 Design Criteria for the Encoding

The encoding is designed to have the following properties, which are especially useful if some arguments are nested arrays:

1. The number of reads necessary to access a value is at most the depth of the value inside the argument array structure, i.e. four reads are needed to retrieve `a_i[k][l][r]`. In a previous version of the ABI, the number of reads scaled linearly with the total number of dynamic parameters in the worst case.
2. The data of a variable or array element is not interleaved with other data and it is relocatable, i.e. it only uses relative "addresses".

2.9.6 Formal Specification of the Encoding

We distinguish static and dynamic types. Static types are encoded in-place and dynamic types are encoded at a separately allocated location after the current block.

Definition: The following types are called "dynamic":

- `bytes`
- `string`
- `T[]` for any `T`
- `T[k]` for any dynamic `T` and any `k >= 0`
- `(T1,...,Tk)` if `Ti` is dynamic for some `1 <= i <= k`

All other types are called "static".

Definition: `len(a)` is the number of bytes in a binary string `a`. The type of `len(a)` is assumed to be `uint256`.

We define `enc`, the actual encoding, as a mapping of values of the ABI types to binary strings such that `len(enc(X))` depends on the value of `X` if and only if the type of `X` is dynamic.

Definition: For any ABI value `X`, we recursively define `enc(X)`, depending on the type of `X` being

- `(T1,...,Tk)` for `k >= 0` and any types `T1`, ..., `Tk`

  `enc(X) = head(X(1)) ... head(X(k)) tail(X(1)) ... tail(X(k))`

  where `X = (X(1), ..., X(k))` and `head` and `tail` are defined for `Ti` as follows:

  if `Ti` is static:

  > `head(X(i)) = enc(X(i))` and `tail(X(i)) = ""` (the empty string)

  otherwise, i.e. if `Ti` is dynamic:

  > `head(X(i)) = enc(len( head(X(1)) ... head(X(k)) tail(X(1)) ... tail(X(i-1)) ))` `tail(X(i)) = enc(X(i))`

  Note that in the dynamic case, `head(X(i))` is well-defined since the lengths of the head parts only depend on the types and not the values. The value of `head(X(i))` is the offset of the beginning of `tail(X(i))` relative to the start of `enc(X)`.

- `T[k]` for any `T` and `k`:

  `enc(X) = enc((X[0], ..., X[k-1]))`

  i.e. it is encoded as if it were a tuple with `k` elements of the same type.

- `T[]` where `X` has `k` elements (`k` is assumed to be of type `uint256`):

  `enc(X) = enc(k) enc([X[0], ..., X[k-1]])`

  i.e. it is encoded as if it were an array of static size `k`, prefixed with the number of elements.

- `bytes`, of length `k` (which is assumed to be of type `uint256`):

  `enc(X) = enc(k) pad_right(X)`, i.e. the number of bytes is encoded as a `uint256` followed by the actual value of `X` as a byte sequence, followed by the minimum number of zero-bytes such that `len(enc(X))` is a multiple of 32.

- `string`:

  `enc(X) = enc(enc_utf8(X))`, i.e. `X` is utf-8 encoded and this value is interpreted as of `bytes` type and encoded further. Note that the length used in this subsequent encoding is the number of bytes of the utf-8 encoded string, not its number of characters.

- uint<M>: enc(X) is the big-endian encoding of X, padded on the higher-order (left) side with zero-bytes such that the length is 32 bytes.
- address: as in the uint160 case
- int<M>: enc(X) is the big-endian two's complement encoding of X, padded on the higher-order (left) side with 0xff for negative X and with zero bytes for positive X such that the length is 32 bytes.
- bool: as in the uint8 case, where 1 is used for true and 0 for false
- fixed<M>x<N>: enc(X) is enc(X * 10**N) where X * 10**N is interpreted as a int256.
- fixed: as in the fixed128x18 case
- ufixed<M>x<N>: enc(X) is enc(X * 10**N) where X * 10**N is interpreted as a uint256.
- ufixed: as in the ufixed128x18 case
- bytes<M>: enc(X) is the sequence of bytes in X padded with trailing zero-bytes to a length of 32 bytes.

Note that for any X, len(enc(X)) is a multiple of 32.

2.9.7 Function Selector and Argument Encoding

All in all, a call to the function f with parameters a_1, ..., a_n is encoded as

```
function_selector(f) enc((a_1, ..., a_n))
```

and the return values v_1, ..., v_k of f are encoded as

```
enc((v_1, ..., v_k))
```

i.e. the values are combined into a tuple and encoded.

2.9.8 Examples

Given the contract:

```
pragma solidity >=0.4.16 <0.7.0;


contract Foo {
    function bar(bytes3[2] memory) public pure {}
    function baz(uint32 x, bool y) public pure returns (bool r) { r = x > 32 || y; }
    function sam(bytes memory, bool, uint[] memory) public pure {}
}
```

Thus for our Foo example if we wanted to call baz with the parameters 69 and true, we would pass 68 bytes total, which can be broken down into:

- 0xcdcd77c0: the Method ID. This is derived as the first 4 bytes of the Keccak hash of the ASCII form of the signature baz(uint32,bool).
- 0x0000000000000000000000000000000000000000000000000000000000000045: the first parameter, a uint32 value 69 padded to 32 bytes
- 0x0000000000000000000000000000000000000000000000000000000000000001: the second parameter - boolean true, padded to 32 bytes

In total:

```
0xcdcd77c0000000000000000000000000000000000000000000000000000000000000004500000000000000000000000
```

It returns a single `bool`. If, for example, it were to return `false`, its output would be the single byte array `0x0000000000000000000000000000000000000000000000000000000000000000`, a single bool.

If we wanted to call `bar` with the argument `["abc", "def"]`, we would pass 68 bytes total, broken down into:

- `0xfce353f6`: the Method ID. This is derived from the signature `bar(bytes3[2])`.
- `0x6162630000000000000000000000000000000000000000000000000000000000`: the first part of the first parameter, a `bytes3` value `"abc"` (left-aligned).
- `0x6465660000000000000000000000000000000000000000000000000000000000`: the second part of the first parameter, a `bytes3` value `"def"` (left-aligned).

In total:

```
0xfce353f66162630000000000000000000000000000000000000000000000000000000000646566000000000000000000
```

If we wanted to call `sam` with the arguments `"dave"`, `true` and `[1,2,3]`, we would pass 292 bytes total, broken down into:

- `0xa5643bf2`: the Method ID. This is derived from the signature `sam(bytes,bool,uint256[])`. Note that `uint` is replaced with its canonical representation `uint256`.
- `0x0000000000000000000000000000000000000000000000000000000000000060`: the location of the data part of the first parameter (dynamic type), measured in bytes from the start of the arguments block. In this case, `0x60`.
- `0x0000000000000000000000000000000000000000000000000000000000000001`: the second parameter: boolean true.
- `0x00000000000000000000000000000000000000000000000000000000000000a0`: the location of the data part of the third parameter (dynamic type), measured in bytes. In this case, `0xa0`.
- `0x0000000000000000000000000000000000000000000000000000000000000004`: the data part of the first argument, it starts with the length of the byte array in elements, in this case, 4.
- `0x6461766500000000000000000000000000000000000000000000000000000000`: the contents of the first argument: the UTF-8 (equal to ASCII in this case) encoding of `"dave"`, padded on the right to 32 bytes.
- `0x0000000000000000000000000000000000000000000000000000000000000003`: the data part of the third argument, it starts with the length of the array in elements, in this case, 3.
- `0x0000000000000000000000000000000000000000000000000000000000000001`: the first entry of the third parameter.
- `0x0000000000000000000000000000000000000000000000000000000000000002`: the second entry of the third parameter.
- `0x0000000000000000000000000000000000000000000000000000000000000003`: the third entry of the third parameter.

In total:

```
0xa5643bf2000000000000000000000000000000000000000000000000000000000000006000000000000000000000000
```

2.9.9 Use of Dynamic Types

A call to a function with the signature `f(uint,uint32[],bytes10,bytes)` with values `(0x123, [0x456, 0x789], "1234567890", "Hello, world!")` is encoded in the following way:

We take the first four bytes of `sha3("f(uint256,uint32[],bytes10,bytes)")`, i.e. `0x8be65246`. Then we encode the head parts of all four arguments. For the static types `uint256` and `bytes10`, these are directly the values we want to pass, whereas for the dynamic types `uint32[]` and `bytes`, we use the offset in bytes to the start of their data area, measured from the start of the value encoding (i.e. not counting the first four bytes containing the hash of the function signature). These are:

- `0x0000000000000000000000000000000000000000000000000000000000000123` (`0x123` padded to 32 bytes)
- `0x0000000000000000000000000000000000000000000000000000000000000080` (offset to start of data part of second parameter, 4*32 bytes, exactly the size of the head part)
- `0x3132333435363738393000000000000000000000000000000000000000000000` (`"1234567890"` padded to 32 bytes on the right)
- `0x00000000000000000000000000000000000000000000000000000000000000e0` (offset to start of data part of fourth parameter = offset to start of data part of first dynamic parameter + size of data part of first dynamic parameter = 4*32 + 3*32 (see below))

After this, the data part of the first dynamic argument, `[0x456, 0x789]` follows:

- `0x0000000000000000000000000000000000000000000000000000000000000002` (number of elements of the array, 2)
- `0x0000000000000000000000000000000000000000000000000000000000000456` (first element)
- `0x0000000000000000000000000000000000000000000000000000000000000789` (second element)

Finally, we encode the data part of the second dynamic argument, `"Hello, world!"`:

- `0x000000000000000000000000000000000000000000000000000000000000000d` (number of elements (bytes in this case): 13)
- `0x48656c6c6f2c20776f726c642100000000000000000000000000000000000000` (`"Hello, world!"` padded to 32 bytes on the right)

All together, the encoding is (newline after function selector and each 32-bytes for clarity):

```
0x8be65246
  0000000000000000000000000000000000000000000000000000000000000123
  0000000000000000000000000000000000000000000000000000000000000080
  3132333435363738393000000000000000000000000000000000000000000000
  00000000000000000000000000000000000000000000000000000000000000e0
  0000000000000000000000000000000000000000000000000000000000000002
  0000000000000000000000000000000000000000000000000000000000000456
  0000000000000000000000000000000000000000000000000000000000000789
  000000000000000000000000000000000000000000000000000000000000000d
  48656c6c6f2c20776f726c642100000000000000000000000000000000000000
```

Let us apply the same principle to encode the data for a function with a signature `g(uint[][],string[])` with values `([[1, 2], [3]], ["one", "two", "three"])` but start from the most atomic parts of the encoding:

First we encode the length and data of the first embedded dynamic array `[1, 2]` of the first root array `[[1, 2], [3]]`:

- 0x0000000000000000000000000000000000000000000000000000000000000002 (number of elements in the first array, 2; the elements themselves are 1 and 2)
- 0x0000000000000000000000000000000000000000000000000000000000000001 (first element)
- 0x0000000000000000000000000000000000000000000000000000000000000002 (second element)

Then we encode the length and data of the second embedded dynamic array [3] of the first root array [[1, 2], [3]]:

- 0x0000000000000000000000000000000000000000000000000000000000000001 (number of elements in the second array, 1; the element is 3)
- 0x0000000000000000000000000000000000000000000000000000000000000003 (first element)

Then we need to find the offsets a and b for their respective dynamic arrays [1, 2] and [3]. To calculate the offsets we can take a look at the encoded data of the first root array [[1, 2], [3]] enumerating each line in the encoding:

```
0 - a                                                                - offset of [1, 2]
1 - b                                                                - offset of [3]
2 - 0000000000000000000000000000000000000000000000000000000000000002 - count for [1, 2]
3 - 0000000000000000000000000000000000000000000000000000000000000001 - encoding of 1
4 - 0000000000000000000000000000000000000000000000000000000000000002 - encoding of 2
5 - 0000000000000000000000000000000000000000000000000000000000000001 - count for [3]
6 - 0000000000000000000000000000000000000000000000000000000000000003 - encoding of 3
```

Offset a points to the start of the content of the array [1, 2] which is line 2 (64 bytes); thus a = 0x0000000000000000000000000000000000000000000000000000000000000040.

Offset b points to the start of the content of the array [3] which is line 5 (160 bytes); thus b = 0x00000000000000000000000000000000000000000000000000000000000000a0.

Then we encode the embedded strings of the second root array:

- 0x0000000000000000000000000000000000000000000000000000000000000003 (number of characters in word "one")
- 0x6f6e650000000000000000000000000000000000000000000000000000000000 (utf8 representation of word "one")
- 0x0000000000000000000000000000000000000000000000000000000000000003 (number of characters in word "two")
- 0x74776f0000000000000000000000000000000000000000000000000000000000 (utf8 representation of word "two")
- 0x0000000000000000000000000000000000000000000000000000000000000005 (number of characters in word "three")
- 0x7468726565000000000000000000000000000000000000000000000000000000 (utf8 representation of word "three")

In parallel to the first root array, since strings are dynamic elements we need to find their offsets c, d and e:

```
0 - c                                                                - offset for "one
↪"
1 - d                                                                - offset for "two
↪"
2 - e                                                                - offset for
↪"three"
3 - 0000000000000000000000000000000000000000000000000000000000000003 - count for "one"
4 - 6f6e650000000000000000000000000000000000000000000000000000000000 - encoding of
↪"one"
5 - 0000000000000000000000000000000000000000000000000000000000000003 - count for "two"
6 - 74776f0000000000000000000000000000000000000000000000000000000000 - encoding of
↪"two"
7 - 0000000000000000000000000000000000000000000000000000000000000005 - count for
↪"three"
8 - 7468726565000000000000000000000000000000000000000000000000000000 - encoding of
↪"three"
```

Offset `c` points to the start of the content of the string `"one"` which is line 3 (96 bytes); thus `c = 0x0000000000000000000000000000000000000000000000000000000000000060`.

Offset `d` points to the start of the content of the string `"two"` which is line 5 (160 bytes); thus `d = 0x00000000000000000000000000000000000000000000000000000000000000a0`.

Offset `e` points to the start of the content of the string `"three"` which is line 7 (224 bytes); thus `e = 0x00000000000000000000000000000000000000000000000000000000000000e0`.

Note that the encodings of the embedded elements of the root arrays are not dependent on each other and have the same encodings for a function with a signature `g(string[],uint[][])`.

Then we encode the length of the first root array:

- `0x0000000000000000000000000000000000000000000000000000000000000002` (number of elements in the first root array, 2; the elements themselves are `[1, 2]` and `[3]`)

Then we encode the length of the second root array:

- `0x0000000000000000000000000000000000000000000000000000000000000003` (number of strings in the second root array, 3; the strings themselves are `"one"`, `"two"` and `"three"`)

Finally we find the offsets `f` and `g` for their respective root dynamic arrays `[[1, 2], [3]]` and `["one", "two", "three"]`, and assemble parts in the correct order:

```
0x2289b18c                                                            - function␣
↪signature
 0 - f                                                                - offset of [[1,
↪ 2], [3]]
 1 - g                                                                - offset of [
↪"one", "two", "three"]
 2 - 0000000000000000000000000000000000000000000000000000000000000002 - count for [[1,
↪ 2], [3]]
 3 - 0000000000000000000000000000000000000000000000000000000000000040 - offset of [1,␣
↪2]
 4 - 00000000000000000000000000000000000000000000000000000000000000a0 - offset of [3]
 5 - 0000000000000000000000000000000000000000000000000000000000000002 - count for [1,␣
↪2]
 6 - 0000000000000000000000000000000000000000000000000000000000000001 - encoding of 1
 7 - 0000000000000000000000000000000000000000000000000000000000000002 - encoding of 2
 8 - 0000000000000000000000000000000000000000000000000000000000000001 - count for [3]
 9 - 0000000000000000000000000000000000000000000000000000000000000003 - encoding of 3
10 - 0000000000000000000000000000000000000000000000000000000000000003 - count for [
↪"one", "two", "three"]
```

(continues on next page)

(continued from previous page)

```
11 - 0000000000000000000000000000000000000000000000000000000000000060 - offset for "one"
12 - 00000000000000000000000000000000000000000000000000000000000000a0 - offset for "two"
13 - 00000000000000000000000000000000000000000000000000000000000000e0 - offset for "three"
14 - 0000000000000000000000000000000000000000000000000000000000000003 - count for "one"
15 - 6f6e650000000000000000000000000000000000000000000000000000000000 - encoding of "one"
16 - 0000000000000000000000000000000000000000000000000000000000000003 - count for "two"
17 - 74776f0000000000000000000000000000000000000000000000000000000000 - encoding of "two"
18 - 0000000000000000000000000000000000000000000000000000000000000005 - count for "three"
19 - 7468726565000000000000000000000000000000000000000000000000000000 - encoding of "three"
```

Offset `f` points to the start of the content of the array `[[1, 2], [3]]` which is line 2 (64 bytes); thus `f = 0x0000000000000000000000000000000000000000000000000000000000000040`.

Offset `g` points to the start of the content of the array `["one", "two", "three"]` which is line 10 (320 bytes); thus `g = 0x0000000000000000000000000000000000000000000000000000000000000140`.

2.9.10 Events

Events are an abstraction of the Ethereum logging/event-watching protocol. Log entries provide the contract's address, a series of up to four topics and some arbitrary length binary data. Events leverage the existing function ABI in order to interpret this (together with an interface spec) as a properly typed structure.

Given an event name and series of event parameters, we split them into two sub-series: those which are indexed and those which are not. Those which are indexed, which may number up to 3, are used alongside the Keccak hash of the event signature to form the topics of the log entry. Those which are not indexed form the byte array of the event.

In effect, a log entry using this ABI is described as:

- `address`: the address of the contract (intrinsically provided by Ethereum);
- `topics[0]: keccak(EVENT_NAME+"("+EVENT_ARGS.map(canonical_type_of).join(",")+")")` (`canonical_type_of` is a function that simply returns the canonical type of a given argument, e.g. for `uint indexed foo`, it would return `uint256`). If the event is declared as `anonymous` the `topics[0]` is not generated;
- `topics[n]`: `abi_encode(EVENT_INDEXED_ARGS[n - 1])` (`EVENT_INDEXED_ARGS` is the series of `EVENT_ARGS` that are indexed);
- `data`: ABI encoding of `EVENT_NON_INDEXED_ARGS` (`EVENT_NON_INDEXED_ARGS` is the series of `EVENT_ARGS` that are not indexed, `abi_encode` is the ABI encoding function used for returning a series of typed values from a function, as described above).

For all types of length at most 32 bytes, the `EVENT_INDEXED_ARGS` array contains the value directly, padded or sign-extended (for signed integers) to 32 bytes, just as for regular ABI encoding. However, for all "complex" types or types of dynamic length, including all arrays, `string`, `bytes` and structs, `EVENT_INDEXED_ARGS` will contain the *Keccak hash* of a special in-place encoded value (see *Encoding of Indexed Event Parameters*), rather than the encoded value directly. This allows applications to efficiently query for values of dynamic-length types (by setting the hash of the encoded value as the topic), but leaves applications unable to decode indexed values they have not queried for. For dynamic-length types, application developers face a trade-off between fast search for predetermined values (if the argument is indexed) and legibility of arbitrary values (which requires that the arguments not be indexed). Developers may overcome this tradeoff and achieve both efficient search and arbitrary legibility by defining events with two arguments — one indexed, one not — intended to hold the same value.

2.9.11 JSON

The JSON format for a contract's interface is given by an array of function and/or event descriptions. A function description is a JSON object with the fields:

- `type`: `"function"`, `"constructor"`, `"receive"` (the *"receive Ether" function*) or `"fallback"` (the *"default" function*);
- `name`: the name of the function;
- `inputs`: an array of objects, each of which contains:
  - `name`: the name of the parameter.
  - `type`: the canonical type of the parameter (more below).
  - `components`: used for tuple types (more below).
- `outputs`: an array of objects similar to `inputs`.
- `stateMutability`: a string with one of the following values: `pure` (*specified to not read blockchain state*), `view` (*specified to not modify the blockchain state*), `nonpayable` (function does not accept Ether) and `payable` (function accepts Ether).

Constructor and fallback function never have `name` or `outputs`. Fallback function doesn't have `inputs` either.

Note: Sending non-zero Ether to non-payable function will revert the transaction.

An event description is a JSON object with fairly similar fields:

- `type`: always `"event"`
- `name`: the name of the event.
- `inputs`: an array of objects, each of which contains:
  - `name`: the name of the parameter.
  - `type`: the canonical type of the parameter (more below).
  - `components`: used for tuple types (more below).
  - `indexed`: `true` if the field is part of the log's topics, `false` if it one of the log's data segment.
- `anonymous`: `true` if the event was declared as `anonymous`.

For example,

```
pragma solidity >=0.5.0 <0.7.0;


contract Test {
    constructor() public { b = hex"12345678901234567890123456789012"; }
    event Event(uint indexed a, bytes32 b);
    event Event2(uint indexed a, bytes32 b);
    function foo(uint a) public { emit Event(a, b); }
```

(continues on next page)

(continued from previous page)

```
    bytes32 b;
}
```

would result in the JSON:

```
[{
"type":"event",
"inputs": [{"name":"a","type":"uint256","indexed":true},{"name":"b","type":"bytes32",
↪"indexed":false}],
"name":"Event"
}, {
"type":"event",
"inputs": [{"name":"a","type":"uint256","indexed":true},{"name":"b","type":"bytes32",
↪"indexed":false}],
"name":"Event2"
}, {
"type":"function",
"inputs": [{"name":"a","type":"uint256"}],
"name":"foo",
"outputs": []
}]
```

Handling tuple types

Despite that names are intentionally not part of the ABI encoding they do make a lot of sense to be included in the JSON to enable displaying it to the end user. The structure is nested in the following way:

An object with members `name`, `type` and potentially `components` describes a typed variable. The canonical type is determined until a tuple type is reached and the string description up to that point is stored in `type` prefix with the word `tuple`, i.e. it will be `tuple` followed by a sequence of `[]` and `[k]` with integers `k`. The components of the tuple are then stored in the member `components`, which is of array type and has the same structure as the top-level object except that `indexed` is not allowed there.

As an example, the code

```
pragma solidity >=0.4.19 <0.7.0;
pragma experimental ABIEncoderV2;


contract Test {
    struct S { uint a; uint[] b; T[] c; }
    struct T { uint x; uint y; }
    function f(S memory s, T memory t, uint a) public {}
    function g() public returns (S memory s, T memory t, uint a) {}
}
```

would result in the JSON:

```
[
  {
    "name": "f",
    "type": "function",
    "inputs": [
      {
        "name": "s",
```

(continues on next page)

(continued from previous page)

```
        "type": "tuple",
        "components": [
          {
            "name": "a",
            "type": "uint256"
          },
          {
            "name": "b",
            "type": "uint256[]"
          },
          {
            "name": "c",
            "type": "tuple[]",
            "components": [
              {
                "name": "x",
                "type": "uint256"
              },
              {
                "name": "y",
                "type": "uint256"
              }
            ]
          }
        ]
      },
      {
        "name": "t",
        "type": "tuple",
        "components": [
          {
            "name": "x",
            "type": "uint256"
          },
          {
            "name": "y",
            "type": "uint256"
          }
        ]
      },
      {
        "name": "a",
        "type": "uint256"
      }
    ],
    "outputs": []
  }
]
```

2.9.12 Strict Encoding Mode

Strict encoding mode is the mode that leads to exactly the same encoding as defined in the formal specification above. This means offsets have to be as small as possible while still not creating overlaps in the data areas and thus no gaps are allowed.

Usually, ABI decoders are written in a straightforward way just following offset pointers, but some decoders might enforce strict mode. The Solidity ABI decoder currently does not enforce strict mode, but the encoder always creates data in strict mode.

2.9.13 Non-standard Packed Mode

Through `abi.encodePacked()`, Solidity supports a non-standard packed mode where:

- types shorter than 32 bytes are neither zero padded nor sign extended and
- dynamic types are encoded in-place and without the length.
- array elements are padded, but still encoded in-place

Furthermore, structs as well as nested arrays are not supported.

As an example, the encoding of `int16(-1), bytes1(0x42), uint16(0x03), string("Hello, world!")` results in:

```
0xffff42000348656c6c6f2c20776f726c6421
  ^^^^                                 int16(-1)
      ^^                               bytes1(0x42)
        ^^^^                           uint16(0x03)
            ^^^^^^^^^^^^^^^^^^^^^^^^^^ string("Hello, world!") without a length field
```

More specifically:

- During the encoding, everything is encoded in-place. This means that there is no distinction between head and tail, as in the ABI encoding, and the length of an array is not encoded.
- The direct arguments of `abi.encodePacked` are encoded without padding, as long as they are not arrays (or `string` or `bytes`).
- The encoding of an array is the concatenation of the encoding of its elements **with** padding.
- Dynamically-sized types like `string`, `bytes` or `uint[]` are encoded without their length field.
- The encoding of `string` or `bytes` does not apply padding at the end unless it is part of an array or struct (then it is padded to a multiple of 32 bytes).

In general, the encoding is ambiguous as soon as there are two dynamically-sized elements, because of the missing length field.

If padding is needed, explicit type conversions can be used: `abi.encodePacked(uint16(0x12)) == hex"0012"`.

Since packed encoding is not used when calling functions, there is no special support for prepending a function selector. Since the encoding is ambiguous, there is no decoding function.

> **Warning:** If you use `keccak256(abi.encodePacked(a, b))` and both `a` and `b` are dynamic types, it is easy to craft collisions in the hash value by moving parts of `a` into `b` and vice-versa. More specifically, `abi.encodePacked("a", "bc") == abi.encodePacked("ab", "c")`. If you use `abi.encodePacked` for signatures, authentication or data integrity, make sure to always use the same types and check that at most one of them is dynamic. Unless there is a compelling reason, `abi.encode` should be preferred.

2.9.14 Encoding of Indexed Event Parameters

Indexed event parameters that are not value types, i.e. arrays and structs are not stored directly but instead a keccak256-hash of an encoding is stored. This encoding is defined as follows:

- the encoding of a `bytes` and `string` value is just the string contents without any padding or length prefix.
- the encoding of a struct is the concatenation of the encoding of its members, always padded to a multiple of 32 bytes (even `bytes` and `string`).
- the encoding of an array (both dynamically- and statically-sized) is the concatenation of the encoding of its elements, always padded to a multiple of 32 bytes (even `bytes` and `string`) and without any length prefix

In the above, as usual, a negative number is padded by sign extension and not zero padded. `bytesNN` types are padded on the right while `uintNN` / `intNN` are padded on the left.

Warning: The encoding of a struct is ambiguous if it contains more than one dynamically-sized array. Because of that, always re-check the event data and do not rely on the search result based on the indexed parameters alone.

2.10 Style Guide

2.10.1 Introduction

This guide is intended to provide coding conventions for writing solidity code. This guide should be thought of as an evolving document that will change over time as useful conventions are found and old conventions are rendered obsolete.

Many projects will implement their own style guides. In the event of conflicts, project specific style guides take precedence.

The structure and many of the recommendations within this style guide were taken from python's pep8 style guide.

The goal of this guide is *not* to be the right way or the best way to write solidity code. The goal of this guide is *consistency*. A quote from python's pep8 captures this concept well.

> A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is most important. But most importantly: know when to be inconsistent – sometimes the style guide just doesn't apply. When in doubt, use your best judgement. Look at other examples and decide what looks best. And don't hesitate to ask!

2.10.2 Code Layout

Indentation

Use 4 spaces per indentation level.

Tabs or Spaces

Spaces are the preferred indentation method.

Mixing tabs and spaces should be avoided.

Blank Lines

Surround top level declarations in solidity source with two blank lines.

Yes:

```
pragma solidity >=0.4.0 <0.7.0;

contract A {
    // ...
}


contract B {
    // ...
}


contract C {
    // ...
}
```

No:

```
pragma solidity >=0.4.0 <0.7.0;

contract A {
    // ...
}
contract B {
    // ...
}

contract C {
    // ...
}
```

Within a contract surround function declarations with a single blank line.

Blank lines may be omitted between groups of related one-liners (such as stub functions for an abstract contract)

Yes:

```
pragma solidity ^0.5.13;

abstract contract A {
    function spam() public virtual pure;
    function ham() public virtual pure;
}


contract B is A {
    function spam() public pure override {
        // ...
    }

    function ham() public pure override {
        // ...
```

(continues on next page)

(continued from previous page)

```
    }
}
```

No:

```
pragma solidity >=0.4.0 <0.7.0;

abstract contract A {
    function spam() virtual pure public;
    function ham() public virtual pure;
}


contract B is A {
    function spam() public pure override {
        // ...
    }
    function ham() public pure override {
        // ...
    }
}
```

Maximum Line Length

Keeping lines under the PEP 8 recommendation to a maximum of 79 (or 99) characters helps readers easily parse the code.

Wrapped lines should conform to the following guidelines.

1. The first argument should not be attached to the opening parenthesis.
2. One, and only one, indent should be used.
3. Each argument should fall on its own line.
4. The terminating element, `);`, should be placed on the final line by itself.

Function Calls

Yes:

```
thisFunctionCallIsReallyLong(
    longArgument1,
    longArgument2,
    longArgument3
);
```

No:

```
thisFunctionCallIsReallyLong(longArgument1,
                              longArgument2,
                              longArgument3
);

thisFunctionCallIsReallyLong(longArgument1,
    longArgument2,
    longArgument3
```

(continues on next page)

(continued from previous page)

```
);

thisFunctionCallIsReallyLong(
    longArgument1, longArgument2,
    longArgument3
);

thisFunctionCallIsReallyLong(
longArgument1,
longArgument2,
longArgument3
);

thisFunctionCallIsReallyLong(
    longArgument1,
    longArgument2,
    longArgument3);
```

Assignment Statements

Yes:

```
thisIsALongNestedMapping[being][set][to_some_value] = someFunction(
    argument1,
    argument2,
    argument3,
    argument4
);
```

No:

```
thisIsALongNestedMapping[being][set][to_some_value] = someFunction(argument1,
                                                                   argument2,
                                                                   argument3,
                                                                   argument4);
```

Event Definitions and Event Emitters

Yes:

```
event LongAndLotsOfArgs(
    address sender,
    address recipient,
    uint256 publicKey,
    uint256 amount,
    bytes32[] options
);

LongAndLotsOfArgs(
    sender,
    recipient,
    publicKey,
    amount,
    options
);
```

No:

```
event LongAndLotsOfArgs(address sender,
                        address recipient,
                        uint256 publicKey,
                        uint256 amount,
                        bytes32[] options);

LongAndLotsOfArgs(sender,
                  recipient,
                  publicKey,
                  amount,
                  options);
```

Source File Encoding

UTF-8 or ASCII encoding is preferred.

Imports

Import statements should always be placed at the top of the file.

Yes:

```
pragma solidity >=0.4.0 <0.7.0;

import "./Owned.sol";

contract A {
    // ...
}

contract B is Owned {
    // ...
}
```

No:

```
pragma solidity >=0.4.0 <0.7.0;

contract A {
    // ...
}


import "./Owned.sol";


contract B is Owned {
    // ...
}
```

Order of Functions

Ordering helps readers identify which functions they can call and to find the constructor and fallback definitions easier.

Functions should be grouped according to their visibility and ordered:

- constructor
- receive function (if exists)
- fallback function (if exists)
- external
- public
- internal
- private

Within a grouping, place the `view` and `pure` functions last.

Yes:

```
pragma solidity ^0.5.13;

contract A {
    constructor() public {
        // ...
    }

    receive() external payable {
        // ...
    }

    fallback() external {
        // ...
    }

    // External functions
    // ...

    // External functions that are view
    // ...

    // External functions that are pure
    // ...

    // Public functions
    // ...

    // Internal functions
    // ...

    // Private functions
    // ...
}
```

No:

```
pragma solidity >=0.4.0 <0.7.0;

contract A {

    // External functions
    // ...

```

(continues on next page)

(continued from previous page)

```
    fallback() external {
        // ...
    }
    receive() external payable {
        // ...
    }

    // Private functions
    // ...

    // Public functions
    // ...

    constructor() public {
        // ...
    }

    // Internal functions
    // ...
}
```

Whitespace in Expressions

Avoid extraneous whitespace in the following situations:

Immediately inside parenthesis, brackets or braces, with the exception of single line function declarations.

Yes:

```
spam(ham[1], Coin({name: "ham"}));
```

No:

```
spam( ham[ 1 ], Coin( { name: "ham" } ) );
```

Exception:

```
function singleLine() public { spam(); }
```

Immediately before a comma, semicolon:

Yes:

```
function spam(uint i, Coin coin) public;
```

No:

```
function spam(uint i , Coin coin) public ;
```

More than one space around an assignment or other operator to align with another:

Yes:

```
x = 1;
y = 2;
long_variable = 3;
```

No:

```
x             = 1;
y             = 2;
long_variable = 3;
```

Don’t include a whitespace in the receive and fallback functions:

Yes:

```
receive() external payable {
    ...
}

fallback() external {
    ...
}
```

No:

```
receive () external payable {
    ...
}

fallback () external {
    ...
}
```

Control Structures

The braces denoting the body of a contract, library, functions and structs should:

- open on the same line as the declaration
- close on their own line at the same indentation level as the beginning of the declaration.
- The opening brace should be preceded by a single space.

Yes:

```
pragma solidity >=0.4.0 <0.7.0;

contract Coin {
    struct Bank {
        address owner;
        uint balance;
    }
}
```

No:

```
pragma solidity >=0.4.0 <0.7.0;

contract Coin
{
    struct Bank {
        address owner;
        uint balance;
```

(continues on next page)

(continued from previous page)

```
    }
}
```

The same recommendations apply to the control structures `if`, `else`, `while`, and `for`.

Additionally there should be a single space between the control structures `if`, `while`, and `for` and the parenthetic block representing the conditional, as well as a single space between the conditional parenthetic block and the opening brace.

Yes:

```
if (...) {
    ...
}

for (...) {
    ...
}
```

No:

```
if (...)
{
    ...
}

while(...){
}

for (...) {
    ...;}
```

For control structures whose body contains a single statement, omitting the braces is ok *if* the statement is contained on a single line.

Yes:

```
if (x < 10)
    x += 1;
```

No:

```
if (x < 10)
    someArray.push(Coin({
        name: 'spam',
        value: 42
    }));
```

For `if` blocks which have an `else` or `else if` clause, the `else` should be placed on the same line as the `if`'s closing brace. This is an exception compared to the rules of other block-like structures.

Yes:

```
if (x < 3) {
    x += 1;
} else if (x > 7) {
    x -= 1;
```

(continues on next page)

(continued from previous page)

```
} else {
    x = 5;
}


if (x < 3)
    x += 1;
else
    x -= 1;
```

No:

```
if (x < 3) {
    x += 1;
}
else {
    x -= 1;
}
```

Function Declaration

For short function declarations, it is recommended for the opening brace of the function body to be kept on the same line as the function declaration.

The closing brace should be at the same indentation level as the function declaration.

The opening brace should be preceded by a single space.

Yes:

```
function increment(uint x) public pure returns (uint) {
    return x + 1;
}

function increment(uint x) public pure onlyowner returns (uint) {
    return x + 1;
}
```

No:

```
function increment(uint x) public pure returns (uint)
{
    return x + 1;
}

function increment(uint x) public pure returns (uint){
    return x + 1;
}

function increment(uint x) public pure returns (uint) {
    return x + 1;
    }

function increment(uint x) public pure returns (uint) {
    return x + 1;}
```

The modifier order for a function should be:

1. Visibility
2. Mutability
3. Virtual
4. Override
5. Custom modifiers

Yes:

```
function balance(uint from) public view override returns (uint)  {
    return balanceOf[from];
}

function kill() public onlyowner {
    selfdestruct(owner);
}
```

No:

```
function balance(uint from) public override view returns (uint)  {
    return balanceOf[from];
}

function kill() onlyowner public {
    selfdestruct(owner);
}
```

For long function declarations, it is recommended to drop each argument onto it's own line at the same indentation level as the function body. The closing parenthesis and opening bracket should be placed on their own line as well at the same indentation level as the function declaration.

Yes:

```
function thisFunctionHasLotsOfArguments(
    address a,
    address b,
    address c,
    address d,
    address e,
    address f
)
    public
{
    doSomething();
}
```

No:

```
function thisFunctionHasLotsOfArguments(address a, address b, address c,
    address d, address e, address f) public {
    doSomething();
}

function thisFunctionHasLotsOfArguments(address a,
                                        address b,
                                        address c,
                                        address d,
```

(continues on next page)

(continued from previous page)

```
                                        address e,
                                        address f) public {
    doSomething();
}

function thisFunctionHasLotsOfArguments(
    address a,
    address b,
    address c,
    address d,
    address e,
    address f) public {
    doSomething();
}
```

If a long function declaration has modifiers, then each modifier should be dropped to its own line.

Yes:

```
function thisFunctionNameIsReallyLong(address x, address y, address z)
    public
    onlyowner
    priced
    returns (address)
{
    doSomething();
}

function thisFunctionNameIsReallyLong(
    address x,
    address y,
    address z,
)
    public
    onlyowner
    priced
    returns (address)
{
    doSomething();
}
```

No:

```
function thisFunctionNameIsReallyLong(address x, address y, address z)
                                      public
                                      onlyowner
                                      priced
                                      returns (address) {
    doSomething();
}

function thisFunctionNameIsReallyLong(address x, address y, address z)
    public onlyowner priced returns (address)
{
    doSomething();
}
```

(continues on next page)

(continued from previous page)

```
function thisFunctionNameIsReallyLong(address x, address y, address z)
    public
    onlyowner
    priced
    returns (address) {
    doSomething();
}
```

Multiline output parameters and return statements should follow the same style recommended for wrapping long lines found in the *Maximum Line Length* section.

Yes:

```
function thisFunctionNameIsReallyLong(
    address a,
    address b,
    address c
)
    public
    returns (
        address someAddressName,
        uint256 LongArgument,
        uint256 Argument
    )
{
    doSomething()

    return (
        veryLongReturnArg1,
        veryLongReturnArg2,
        veryLongReturnArg3
    );
}
```

No:

```
function thisFunctionNameIsReallyLong(
    address a,
    address b,
    address c
)
    public
    returns (address someAddressName,
             uint256 LongArgument,
             uint256 Argument)
{
    doSomething()

    return (veryLongReturnArg1,
            veryLongReturnArg1,
            veryLongReturnArg1);
}
```

For constructor functions on inherited contracts whose bases require arguments, it is recommended to drop the base constructors onto new lines in the same manner as modifiers if the function declaration is long or hard to read.

Yes:

```
pragma solidity >=0.4.0 <0.7.0;

// Base contracts just to make this compile
contract B {
    constructor(uint) public {
    }
}
contract C {
    constructor(uint, uint) public {
    }
}
contract D {
    constructor(uint) public {
    }
}

contract A is B, C, D {
    uint x;

    constructor(uint param1, uint param2, uint param3, uint param4, uint param5)
        B(param1)
        C(param2, param3)
        D(param4)
        public
    {
        // do something with param5
        x = param5;
    }
}
```

No:

```
pragma solidity >=0.4.0 <0.7.0;


// Base contracts just to make this compile
contract B {
    constructor(uint) public {
    }
}


contract C {
    constructor(uint, uint) public {
    }
}


contract D {
    constructor(uint) public {
    }
}


contract A is B, C, D {
    uint x;
```

(continues on next page)

(continued from previous page)

```
    constructor(uint param1, uint param2, uint param3, uint param4, uint param5)
    B(param1)
    C(param2, param3)
    D(param4)
    public {
        x = param5;
    }
}


contract X is B, C, D {
    uint x;

    constructor(uint param1, uint param2, uint param3, uint param4, uint param5)
        B(param1)
        C(param2, param3)
        D(param4)
        public {
            x = param5;
        }
}
```

When declaring short functions with a single statement, it is permissible to do it on a single line.

Permissible:

```
function shortFunction() public { doSomething(); }
```

These guidelines for function declarations are intended to improve readability. Authors should use their best judgement as this guide does not try to cover all possible permutations for function declarations.

Mappings

In variable declarations, do not separate the keyword `mapping` from its type by a space. Do not separate any nested `mapping` keyword from its type by whitespace.

Yes:

```
mapping(uint => uint) map;
mapping(address => bool) registeredAddresses;
mapping(uint => mapping(bool => Data[])) public data;
mapping(uint => mapping(uint => s)) data;
```

No:

```
mapping (uint => uint) map;
mapping( address => bool ) registeredAddresses;
mapping (uint => mapping (bool => Data[])) public data;
mapping(uint => mapping (uint => s)) data;
```

Variable Declarations

Declarations of array variables should not have a space between the type and the brackets.

Yes:

```
uint[] x;
```

No:

```
uint [] x;
```

Other Recommendations

- Strings should be quoted with double-quotes instead of single-quotes.

Yes:

```
str = "foo";
str = "Hamlet says, 'To be or not to be...'";
```

No:

```
str = 'bar';
str = '"Be yourself; everyone else is already taken." -Oscar Wilde';
```

- Surround operators with a single space on either side.

Yes:

```
x = 3;
x = 100 / 10;
x += 3 + 4;
x |= y && z;
```

No:

```
x=3;
x = 100/10;
x += 3+4;
x |= y&&z;
```

- Operators with a higher priority than others can exclude surrounding whitespace in order to denote precedence. This is meant to allow for improved readability for complex statement. You should always use the same amount of whitespace on either side of an operator:

Yes:

```
x = 2**3 + 5;
x = 2*y + 3*z;
x = (a+b) * (a-b);
```

No:

```
x = 2** 3 + 5;
x = y+z;
x +=1;
```

2.10.3 Order of Layout

Layout contract elements in the following order:

1. Pragma statements
2. Import statements
3. Interfaces
4. Libraries
5. Contracts

Inside each contract, library or interface, use the following order:

1. Type declarations
2. State variables
3. Events
4. Functions

Note: It might be clearer to declare types close to their use in events or state variables.

2.10.4 Naming Conventions

Naming conventions are powerful when adopted and used broadly. The use of different conventions can convey significant *meta* information that would otherwise not be immediately available.

The naming recommendations given here are intended to improve the readability, and thus they are not rules, but rather guidelines to try and help convey the most information through the names of things.

Lastly, consistency within a codebase should always supersede any conventions outlined in this document.

Naming Styles

To avoid confusion, the following names will be used to refer to different naming styles.

- `b` (single lowercase letter)
- `B` (single uppercase letter)
- `lowercase`
- `lower_case_with_underscores`
- `UPPERCASE`
- `UPPER_CASE_WITH_UNDERSCORES`
- `CapitalizedWords` (or CapWords)
- `mixedCase` (differs from CapitalizedWords by initial lowercase character!)
- `Capitalized_Words_With_Underscores`

Note: When using initialisms in CapWords, capitalize all the letters of the initialisms. Thus HTTPServerError is better than HttpServerError. When using initialisms in mixedCase, capitalize all the letters of the initialisms, except keep the first one lower case if it is the beginning of the name. Thus xmlHTTPRequest is better than XMLHTTPRequest.

Names to Avoid

- `l` - Lowercase letter el
- `O` - Uppercase letter oh
- `I` - Uppercase letter eye

Never use any of these for single letter variable names. They are often indistinguishable from the numerals one and zero.

Contract and Library Names

- Contracts and libraries should be named using the CapWords style. Examples: `SimpleToken`, `SmartBank`, `CertificateHashRepository`, `Player`, `Congress`, `Owned`.
- Contract and library names should also match their filenames.
- If a contract file includes multiple contracts and/or libraries, then the filename should match the *core contract*. This is not recommended however if it can be avoided.

As shown in the example below, if the contract name is *Congress* and the library name is *Owned*, then their associated filenames should be *Congress.sol* and *Owned.sol*.

Yes:

```
pragma solidity >=0.4.0 <0.7.0;


// Owned.sol
contract Owned {
    address public owner;

    constructor() public {
        owner = msg.sender;
    }

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    function transferOwnership(address newOwner) public onlyOwner {
        owner = newOwner;
    }
}
```

and in `Congress.sol`:

```
pragma solidity >=0.4.0 <0.7.0;

import "./Owned.sol";


contract Congress is Owned, TokenRecipient {
    //...
}
```

No:

```
pragma solidity >=0.4.0 <0.7.0;


// owned.sol
contract owned {
    address public owner;

    constructor() public {
        owner = msg.sender;
    }

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    function transferOwnership(address newOwner) public onlyOwner {
        owner = newOwner;
    }
}
```

and in `Congress.sol`:

```
import "./owned.sol";


contract Congress is owned, tokenRecipient {
    //...
}
```

Struct Names

Structs should be named using the CapWords style. Examples: `MyCoin`, `Position`, `PositionXY`.

Event Names

Events should be named using the CapWords style. Examples: `Deposit`, `Transfer`, `Approval`, `BeforeTransfer`, `AfterTransfer`.

Function Names

Functions other than constructors should use mixedCase. Examples: `getBalance`, `transfer`, `verifyOwner`, `addMember`, `changeOwner`.

Function Argument Names

Function arguments should use mixedCase. Examples: `initialSupply`, `account`, `recipientAddress`, `senderAddress`, `newOwner`.

When writing library functions that operate on a custom struct, the struct should be the first argument and should always be named `self`.

Local and State Variable Names

Use mixedCase. Examples: `totalSupply`, `remainingSupply`, `balancesOf`, `creatorAddress`, `isPreSale`, `tokenExchangeRate`.

Constants

Constants should be named with all capital letters with underscores separating words. Examples: `MAX_BLOCKS`, `TOKEN_NAME`, `TOKEN_TICKER`, `CONTRACT_VERSION`.

Modifier Names

Use mixedCase. Examples: `onlyBy`, `onlyAfter`, `onlyDuringThePreSale`.

Enums

Enums, in the style of simple type declarations, should be named using the CapWords style. Examples: `TokenGroup`, `Frame`, `HashStyle`, `CharacterLocation`.

Avoiding Naming Collisions

- `single_trailing_underscore_`

This convention is suggested when the desired name collides with that of a built-in or otherwise reserved name.

2.10.5 NatSpec

Solidity contracts can have a form of comments that are the basis of the Ethereum Natural Language Specification Format.

Add comments above functions or contracts following doxygen notation of one or multiple lines starting with /// or a multiline comment starting with /** and ending with */.

For example, the contract from a simple smart contract with the comments added looks like the one below:

```
pragma solidity >=0.4.0 <0.7.0;


/// @author The Solidity Team
/// @title A simple storage example
contract SimpleStorage {
    uint storedData;

    /// Store `x`.
    /// @param x the new value to store
    /// @dev stores the number in the state variable `storedData`
    function set(uint x) public {
        storedData = x;
    }

    /// Return the stored value.
    /// @dev retrieves the value of the state variable `storedData`
    /// @return the stored value
```

(continues on next page)

(continued from previous page)

```
    function get() public view returns (uint) {
        return storedData;
    }
}
```

It is recommended that Solidity contracts are fully annotated using NatSpec for all public interfaces (everything in the ABI).

Please see the section about NatSpec for a detailed explanation.

2.11 Common Patterns

2.11.1 Withdrawal from Contracts

The recommended method of sending funds after an effect is using the withdrawal pattern. Although the most intuitive method of sending lat, as a result of an effect, is a direct `transfer` call, this is not recommended as it introduces a potential security risk. You may read more about this on the security_considerations page.

The following is an example of the withdrawal pattern in practice in a contract where the goal is to send the most money to the contract in order to become the "richest".

In the following contract, if you are no longer the richest, you receive the funds of the person who is now the richest.

```
pragma solidity >=0.5.0 <0.7.0;

contract WithdrawalContract {
    address public richest;
    uint public mostSent;

    mapping (address => uint) pendingWithdrawals;

    constructor() public payable {
        richest = msg.sender;
        mostSent = msg.value;
    }

    function becomeRichest() public payable {
        require(msg.value > mostSent, "Not enough money sent.");
        pendingWithdrawals[richest] += msg.value;
        richest = msg.sender;
        mostSent = msg.value;
    }

    function withdraw() public {
        uint amount = pendingWithdrawals[msg.sender];
        // Remember to zero the pending refund before
        // sending to prevent re-entrancy attacks
        pendingWithdrawals[msg.sender] = 0;
        msg.sender.transfer(amount);
    }
}
```

This is as opposed to the more intuitive sending pattern:

```
pragma solidity >=0.5.0 <0.7.0;

contract SendContract {
    address payable public richest;
    uint public mostSent;

    constructor() public payable {
        richest = msg.sender;
        mostSent = msg.value;
    }

    function becomeRichest() public payable {
        require(msg.value > mostSent, "Not enough money sent.");
        // This line can cause problems (explained below).
        richest.transfer(msg.value);
        richest = msg.sender;
        mostSent = msg.value;
    }
}
```

Notice that, in this example, an attacker could trap the contract into an unusable state by causing `richest` to be the address of a contract that has a receive or fallback function which fails (e.g. by using `revert()` or by just consuming more than the 2300 gas stipend transferred to them). That way, whenever `transfer` is called to deliver funds to the "poisoned" contract, it will fail and thus also `becomeRichest` will fail, with the contract being stuck forever.

In contrast, if you use the "withdraw" pattern from the first example, the attacker can only cause his or her own withdraw to fail and not the rest of the contract's workings.

2.11.2 Restricting Access

Restricting access is a common pattern for contracts. Note that you can never restrict any human or computer from reading the content of your transactions or your contract's state. You can make it a bit harder by using encryption, but if your contract is supposed to read the data, so will everyone else.

You can restrict read access to your contract's state by **other contracts**. That is actually the default unless you declare your state variables `public`.

Furthermore, you can restrict who can make modifications to your contract's state or call your contract's functions and this is what this section is about.

The use of **function modifiers** makes these restrictions highly readable.

```
pragma solidity >=0.4.22 <0.7.0;

contract AccessRestriction {
    // These will be assigned at the construction
    // phase, where `msg.sender` is the account
    // creating this contract.
    address public owner = msg.sender;
    uint public creationTime = now;

    // Modifiers can be used to change
    // the body of a function.
    // If this modifier is used, it will
    // prepend a check that only passes
    // if the function is called from
    // a certain address.
```

(continues on next page)

(continued from previous page)

```
    modifier onlyBy(address _account)
    {
        require(
            msg.sender == _account,
            "Sender not authorized."
        );
        // Do not forget the "_;"! It will
        // be replaced by the actual function
        // body when the modifier is used.
        _;
    }

    /// Make `_newOwner` the new owner of this
    /// contract.
    function changeOwner(address _newOwner)
        public
        onlyBy(owner)
    {
        owner = _newOwner;
    }

    modifier onlyAfter(uint _time) {
        require(
            now >= _time,
            "Function called too early."
        );
        _;
    }

    /// Erase ownership information.
    /// May only be called 6 weeks after
    /// the contract has been created.
    function disown()
        public
        onlyBy(owner)
        onlyAfter(creationTime + 6 weeks)
    {
        delete owner;
    }

    // This modifier requires a certain
    // fee being associated with a function call.
    // If the caller sent too much, he or she is
    // refunded, but only after the function body.
    // This was dangerous before Solidity version 0.4.0,
    // where it was possible to skip the part after `_;`.
    modifier costs(uint _amount) {
        require(
            msg.value >= _amount,
            "Not enough lat provided."
        );
        _;
        if (msg.value > _amount)
            msg.sender.transfer(msg.value - _amount);
    }

    function forceOwnerChange(address _newOwner)
```

(continues on next page)

(continued from previous page)

```
        public
        payable
        costs(200 lat)
    {
        owner = _newOwner;
        // just some example condition
        if (uint(owner) & 0 == 1)
            // This did not refund for Solidity
            // before version 0.4.0.
            return;
        // refund overpaid fees
    }
}
```

A more specialised way in which access to function calls can be restricted will be discussed in the next example.

2.11.3 State Machine

Contracts often act as a state machine, which means that they have certain **stages** in which they behave differently or in which different functions can be called. A function call often ends a stage and transitions the contract into the next stage (especially if the contract models **interaction**). It is also common that some stages are automatically reached at a certain point in **time**.

An example for this is a blind auction contract which starts in the stage "accepting blinded bids", then transitions to "revealing bids" which is ended by "determine auction outcome".

Function modifiers can be used in this situation to model the states and guard against incorrect usage of the contract.

Example

In the following example, the modifier `atStage` ensures that the function can only be called at a certain stage.

Automatic timed transitions are handled by the modifier `timeTransitions`, which should be used for all functions.

Note: Modifier Order Matters. If atStage is combined with timedTransitions, make sure that you mention it after the latter, so that the new stage is taken into account.

Finally, the modifier `transitionNext` can be used to automatically go to the next stage when the function finishes.

Note: Modifier May be Skipped. This only applies to Solidity before version 0.4.0: Since modifiers are applied by simply replacing code and not by using a function call, the code in the transitionNext modifier can be skipped if the function itself uses return. If you want to do that, make sure to call nextStage manually from those functions. Starting with version 0.4.0, modifier code will run even if the function explicitly returns.

```
pragma solidity >=0.4.22 <0.7.0;

contract StateMachine {
    enum Stages {
        AcceptingBlindedBids,
        RevealBids,
        AnotherStage,
```

(continues on next page)

(continued from previous page)

```
        AreWeDoneYet,
        Finished
    }

    // This is the current stage.
    Stages public stage = Stages.AcceptingBlindedBids;

    uint public creationTime = now;

    modifier atStage(Stages _stage) {
        require(
            stage == _stage,
            "Function cannot be called at this time."
        );
        _;
    }

    function nextStage() internal {
        stage = Stages(uint(stage) + 1);
    }

    // Perform timed transitions. Be sure to mention
    // this modifier first, otherwise the guards
    // will not take the new stage into account.
    modifier timedTransitions() {
        if (stage == Stages.AcceptingBlindedBids &&
                    now >= creationTime + 10 days)
            nextStage();
        if (stage == Stages.RevealBids &&
                now >= creationTime + 12 days)
            nextStage();
        // The other stages transition by transaction
        _;
    }

    // Order of the modifiers matters here!
    function bid()
        public
        payable
        timedTransitions
        atStage(Stages.AcceptingBlindedBids)
    {
        // We will not implement that here
    }

    function reveal()
        public
        timedTransitions
        atStage(Stages.RevealBids)
    {
    }

    // This modifier goes to the next stage
    // after the function is done.
    modifier transitionNext()
    {
        _;
```

(continues on next page)

(continued from previous page)

```
        nextStage();
    }

    function g()
        public
        timedTransitions
        atStage(Stages.AnotherStage)
        transitionNext
    {
    }

    function h()
        public
        timedTransitions
        atStage(Stages.AreWeDoneYet)
        transitionNext
    {
    }

    function i()
        public
        timedTransitions
        atStage(Stages.Finished)
    {
    }
}
```

2.12 LLL

LLL is a low-level language for the EVM with an s-expressions syntax.

The Solidity repository contains an LLL compiler, which shares the assembler subsystem with Solidity. However, apart from maintaining that it still compiles, no other improvements are made to it.

It is not built unless specifically requested:

```
$ cmake -DLLL=ON ..
$ cmake --build .
```

> **Warning:** The LLL codebase is deprecated and will be removed from the Solidity repository in the future.

Index