**Introduction to Solidity**

**Table of Contents**

* Introduction to Smart Contracts
* Remix IDE
* Solidity Installation
* Installing Solidity in npm / Node.js
* Layout of a Solidity Source File
* SPDX Licence Identifier
* Pragmas
* Comments in Solidity
* Structure of a Smart Contract

**Introduction to Smart Contracts**

A smart contract is a computer software stored on the Blockchain that automatically executes and enforces an agreement. A smart contract is a computer programme written in a language that a computer or target machine can comprehend.

It also includes business logic agreements between parties. Smart contracts also run automatically when specific criteria are satisfied. They are enforceable, meaning that even in the face of opponents, all contractual provisions are implemented as stipulated.

Enforcement is a wide phrase that includes both conventional legal enforcement and the deployment of specialized methods and controls that allow for contract execution without mediation. True smart contracts should not depend on conventional enforcement mechanisms. This means there is no need for an arbitrator or a third party to control or influence the smart contract execution. Smart contracts are self-enforcing, not legally binding.

They must also be secure and unstoppable, which means they must be fault-tolerant and executable in a reasonable length of time. If external variables are adverse, these programs should be able to run and maintain a healthy state. Imagine a conventional computer programme that has logic and executes the instructions it contains. If the program's environment or external circumstances differ from the predicted state, the programme may respond randomly or abort. Smart contracts must be resistant to this sort of problem.

Secure and unstoppable traits may be regarded as essential or desired features, and including them in the smart contract specification from the start would bring more long-term advantages. This will enable researchers to concentrate on these elements from the outset, laying solid foundations for future study. Specific academics believe smart contracts do not need to be automatically executable, but rather automatable, owing to the requirement for manual human input in some cases.

A certified medical expert may be needed to manually verify a medical record. Automated solutions may not be the best option. While human input and control is useful in certain instances, it is not required; a smart contract must be totally automated.

A state machine concept is used to manage smart contracts' internal state. This enables the creation of a programming framework for smart contracts, where the state of a contract is advanced depending on preset criteria.

Although smart contracts express and enforce all contractual provisions, a court of law does not comprehend the code. This dilemma presents various legal concerns concerning smart contracts: Can it be created in a manner that a court would accept and understand? Is it possible to incorporate dispute resolution inside the code? Before smart contracts are utilized as effectively as conventional legal agreements, regulatory and compliance issues must also be addressed.

Despite their name, smart contracts perform exactly what they are intended to do, which is excellent since this characteristic assures that smart contracts provide the same outcome every time they are run. Due to the persistent consensus needs of blockchain systems, smart contracts are deterministic. So smart contracts are just doing what they are supposed to do.

This creates a wide divide between the actual world and the blockchain realm. In this case, the smart contract cannot comprehend plain language, and the code cannot grasp the natural world. So, how can real-life contracts be put on a blockchain? How can this real-world-to-smart-contract bridge be built?

The above questions provide for making smart contract code intelligible by both computers and humans. Smart contracts may be more acceptable in legal contexts if they are written in code that both humans and computers can comprehend. This desired attribute has been extensively researched to solve problems about contract semantics, meaning, and interpretation.

Smart contracts must be deterministic by definition. This characteristic allows any network node to execute a smart contract and obtain the same outcome. If the outcome varies marginally across nodes, then the distributed consensus paradigm on blockchain may fail. The contract language itself should be deterministic to ensure the smart contracts' integrity and stability. The language is deterministic since it has no non-deterministic functions that may yield different outputs on different nodes.

Deterministic smart contracts deliver the same outcome for a given input. That is, when programmes are performed, they provide dependable and accurate business logic that meets all criteria defined in high-level code.

A smart contract has four properties:

**Automatically executable**

**Immutable**

**Enforceable**

**Deterministic**

**Semantically sound**

**Unbreakable**

The first four qualities must be met, whereas the final two may not be needed or feasible in certain cases. For example, a financial derivatives contract should be instantly executable and enforceable at a minimum.

**Solidity Installation**

Solidity compiler can be installed using one of the following ways-

1. **Using an open source in-browser IDE called as Remix -**

You may access Remix online without installing anything. If you wish to utilize it offline, go to — https://github.com/ethereum/remix-live/tree/gh-pages and download the .zip file. There is No need to install several Solidity versions while using Remix.

1. **Using NPM/Node.js -**

Installing Solc-js, a Solidity compiler, is easy and portable using NPM. Solc-js gives Solidity compiler Javascript bindings.

The solc-js project uses the same compiler source code as the C++ solc project. Solc-js is directly usable in JavaScript (such as Remix).

**Remix IDE**

Remix IDE is an open source online and desktop application. A comprehensive selection of plugins with intuitive GUIs helps speed up development of Smart Contracts in Remix . Remix is used for contract development as well as studying and teaching Ethereum.

Remix IDE is part of the Remix Project, a framework for plugin-based development tools. It includes Remix Plugin Engine, Remix Libs, and Remix-IDE.

Solidity contracts may be written using Remix IDE, a sophisticated open source tool.

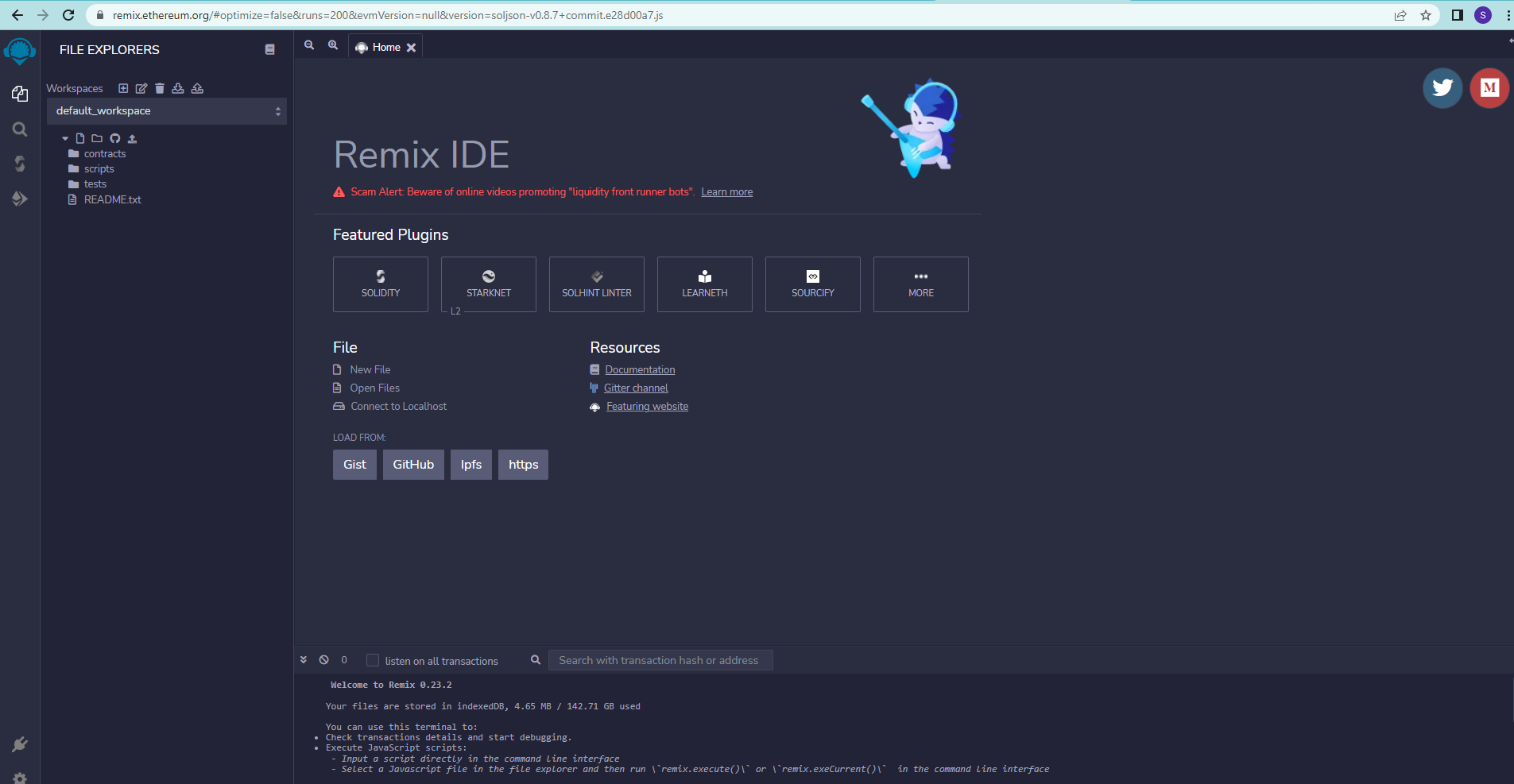
It's developed in JavaScript and works in both the browser and a desktop version.

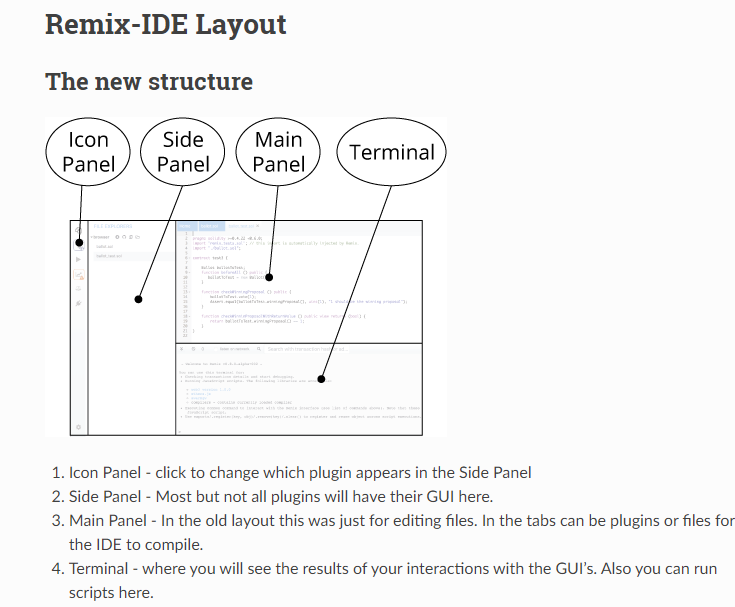
Remix IDE offers components for smart contract testing, debugging, and deployment.

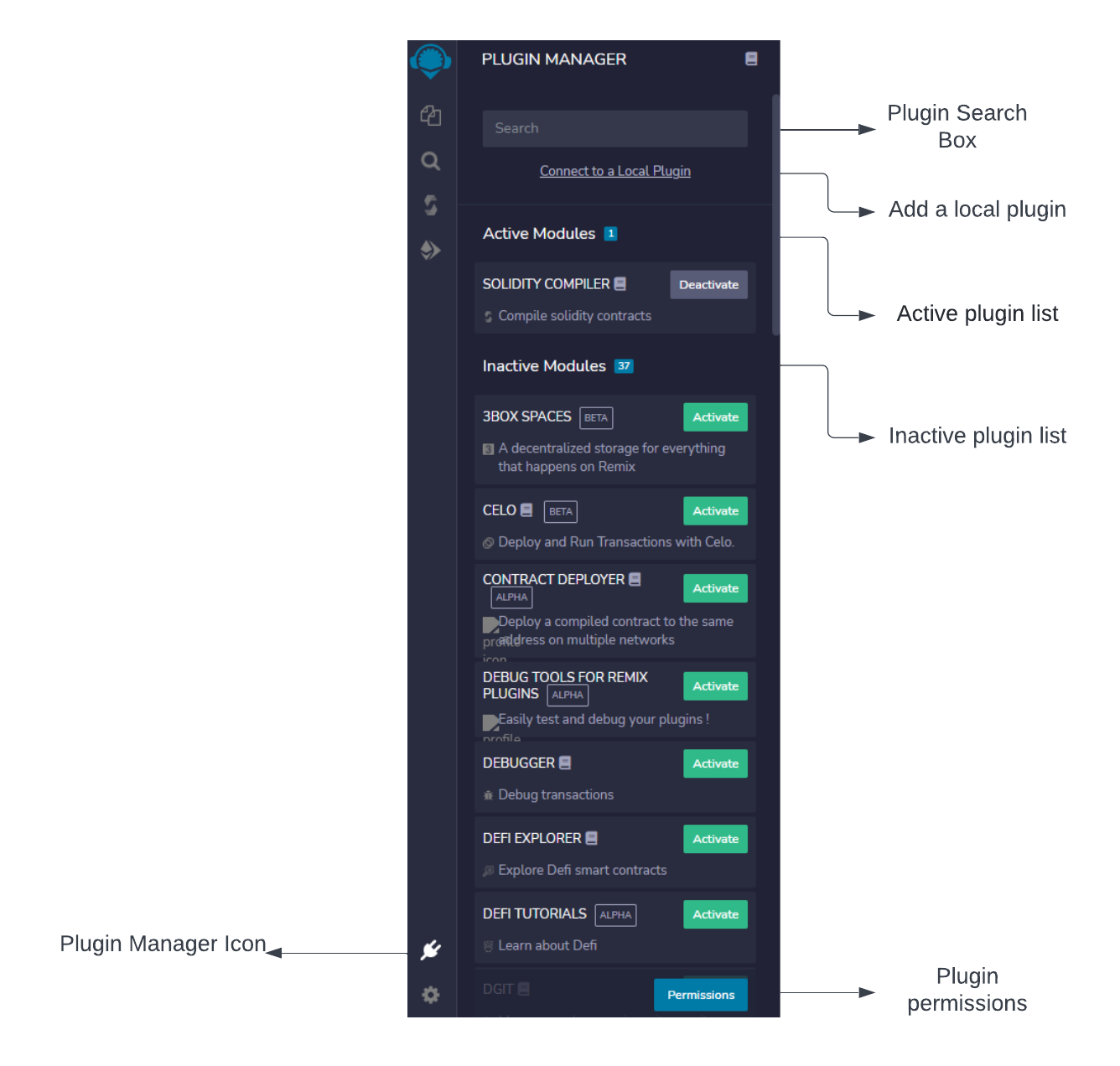
Use the following hyperlink to open the Remix IDE-

[https://remix.ethereum.org](https://remix.ethereum.org/)

Remix Homepage-



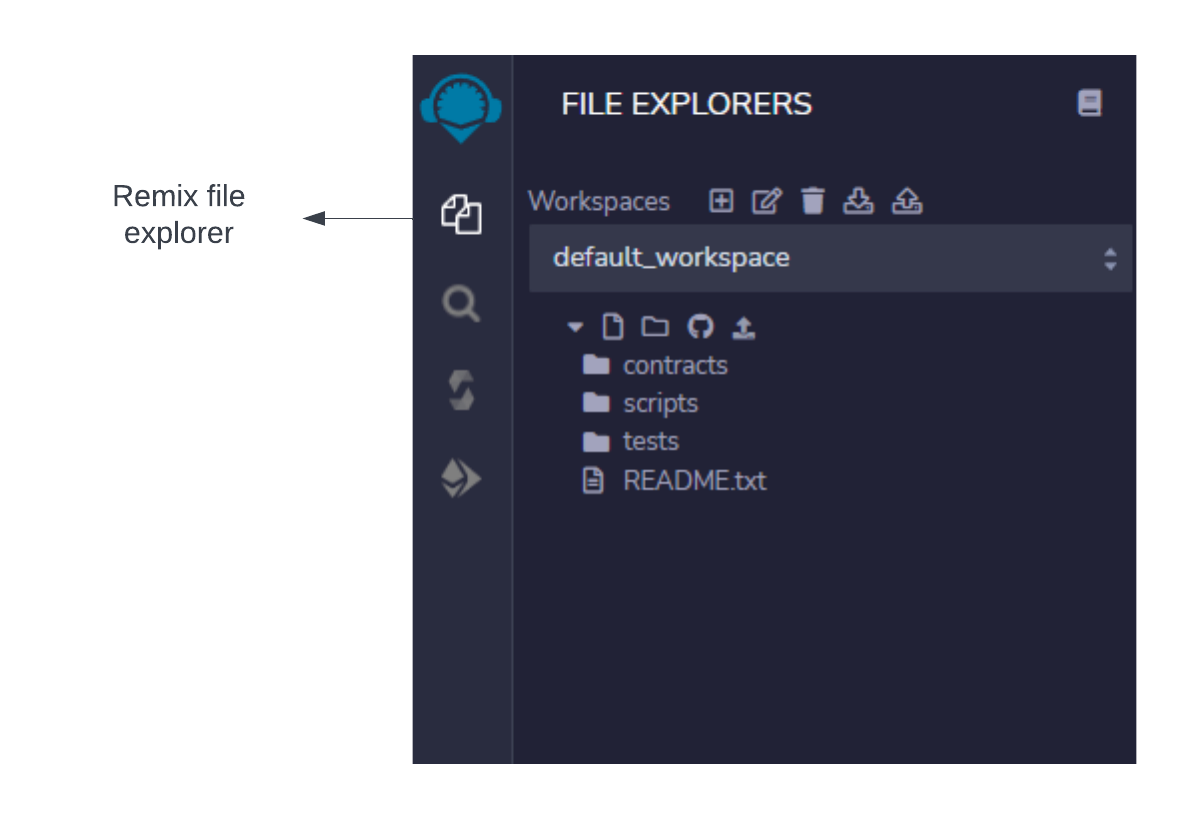
****

Remix Plugin Manager- [](https://lucid.app/documents/edit/e6398182-b862-481c-af30-f1d52d5905e5/0?callback=close&name=docs&callback_type=back&v=368&s=668)

Remix IDE only loads required features. The Plugin Manager is where you control which plugins are activated.

This plugin design allowed the Remix team to add technologies from other teams. Also, Remix or elements of Remix may be merged into other projects.

Remix File Explorer-

[](https://lucid.app/documents/edit/129a2480-3c29-43de-bbe0-a19629d9163f/0?callback=close&name=docs&callback_type=back&v=195&s=595.4399999999999)

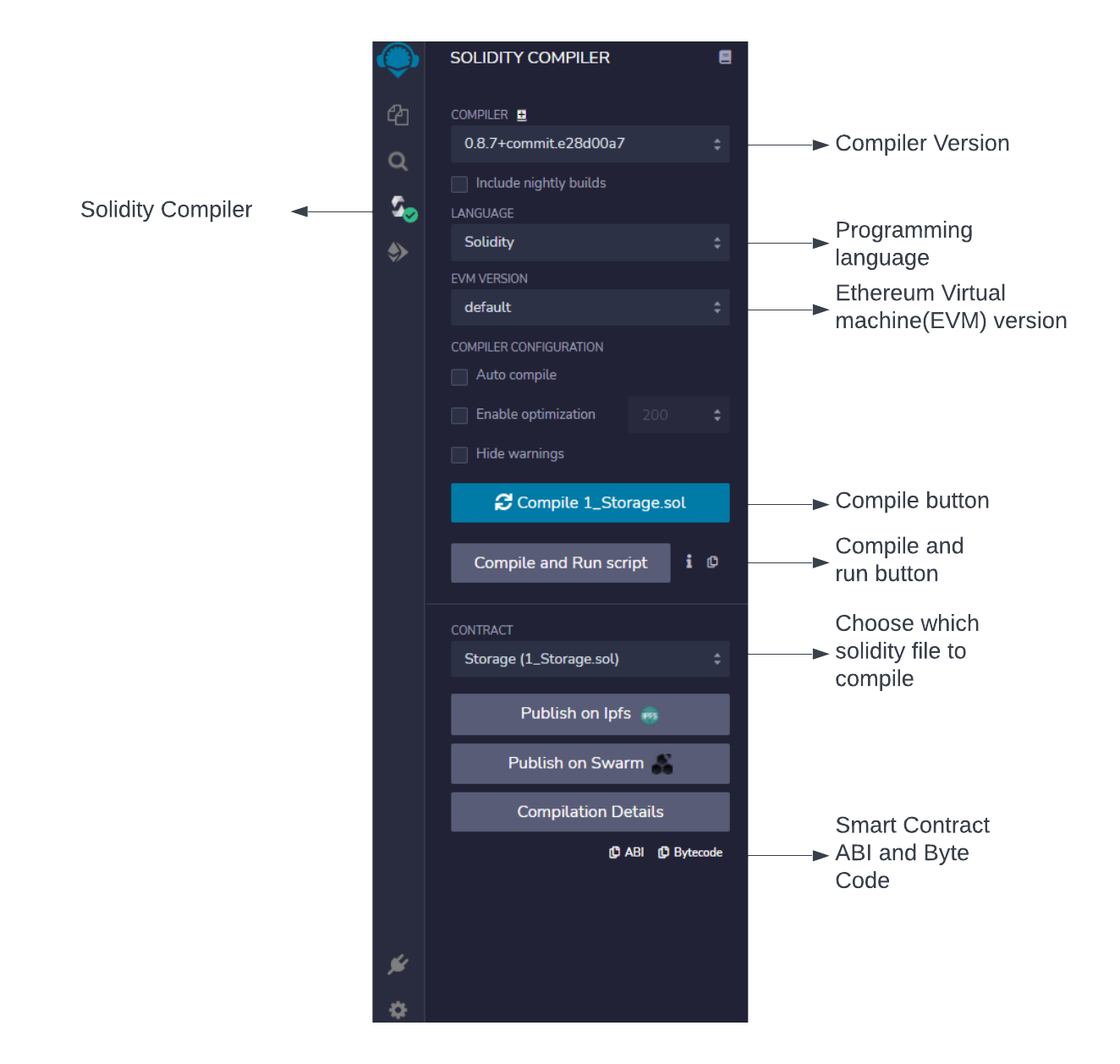
By default, Remix IDE stores files in Workspaces which are folders in your browser’s local storage.

Important Note: Clearing the browser storage will permanently delete all the files stored there.

Important - All the solidity files are stored under the ‘contracts’ folder in the current workspace

To create a new solidity file,add a new file inside the contracts folder with .sol extension.

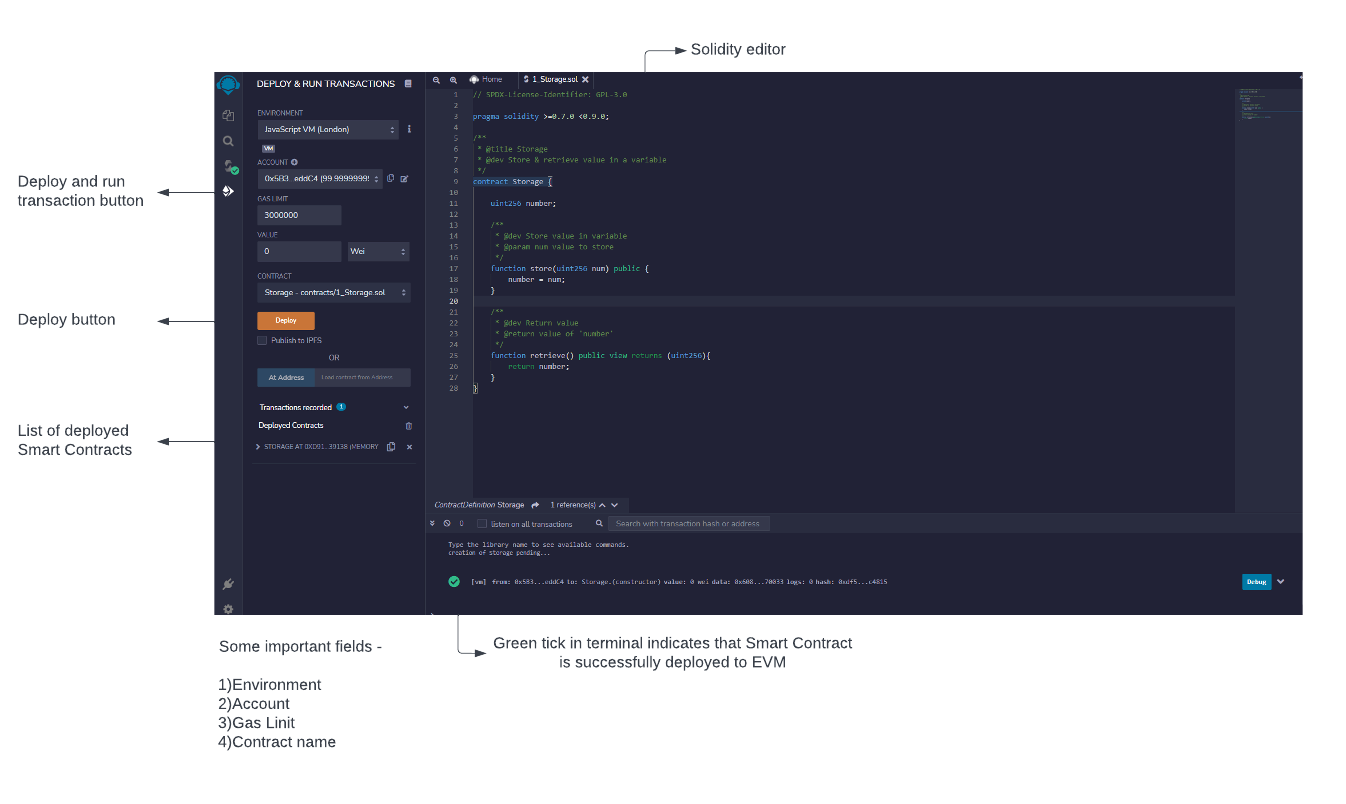
To edit a solidity file , click on the required .sol file.

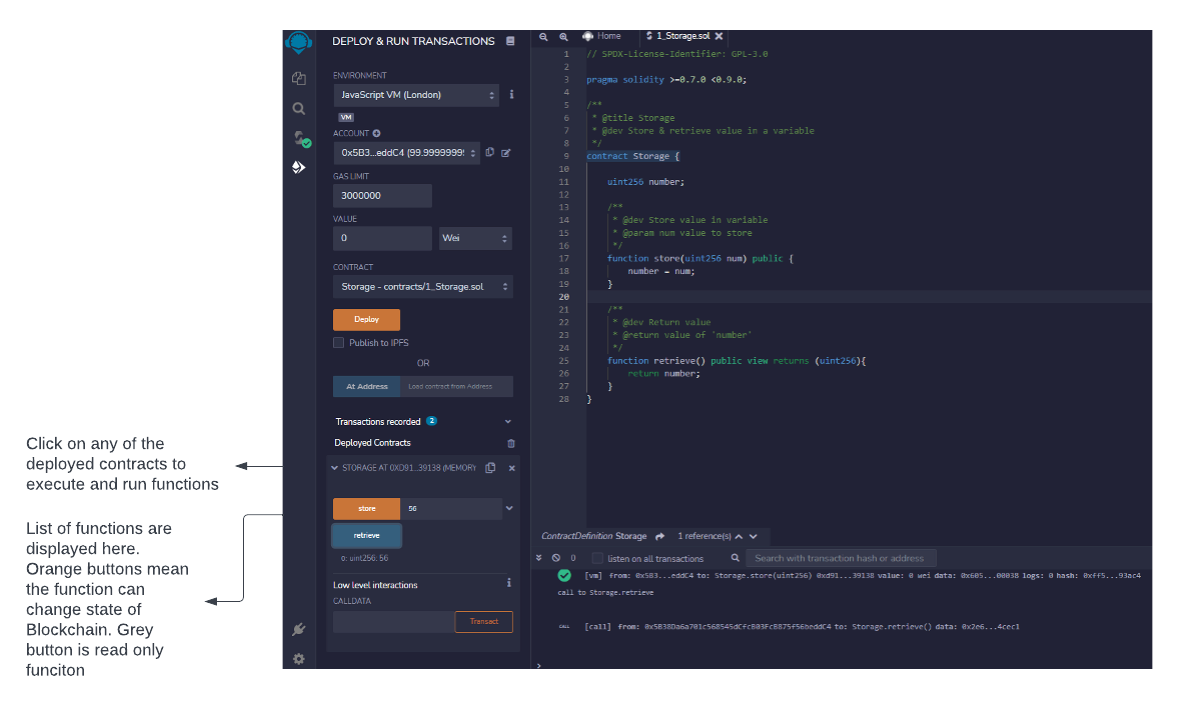
Remix Solidity Compiler-[](https://lucid.app/documents/edit/36786499-7eb2-4aa8-b763-69d565cffe72/0?callback=close&name=docs&callback_type=back&v=485&s=662)

The Solidity Compiler may be accessed from the icon panel by clicking the Solidity icon.

When you click the compile button, the program compiles. If you want the file to be compiled each time the file is saved or when another file is selected - check the auto compile checkbox

Deploying Smart Contracts and running transactions-

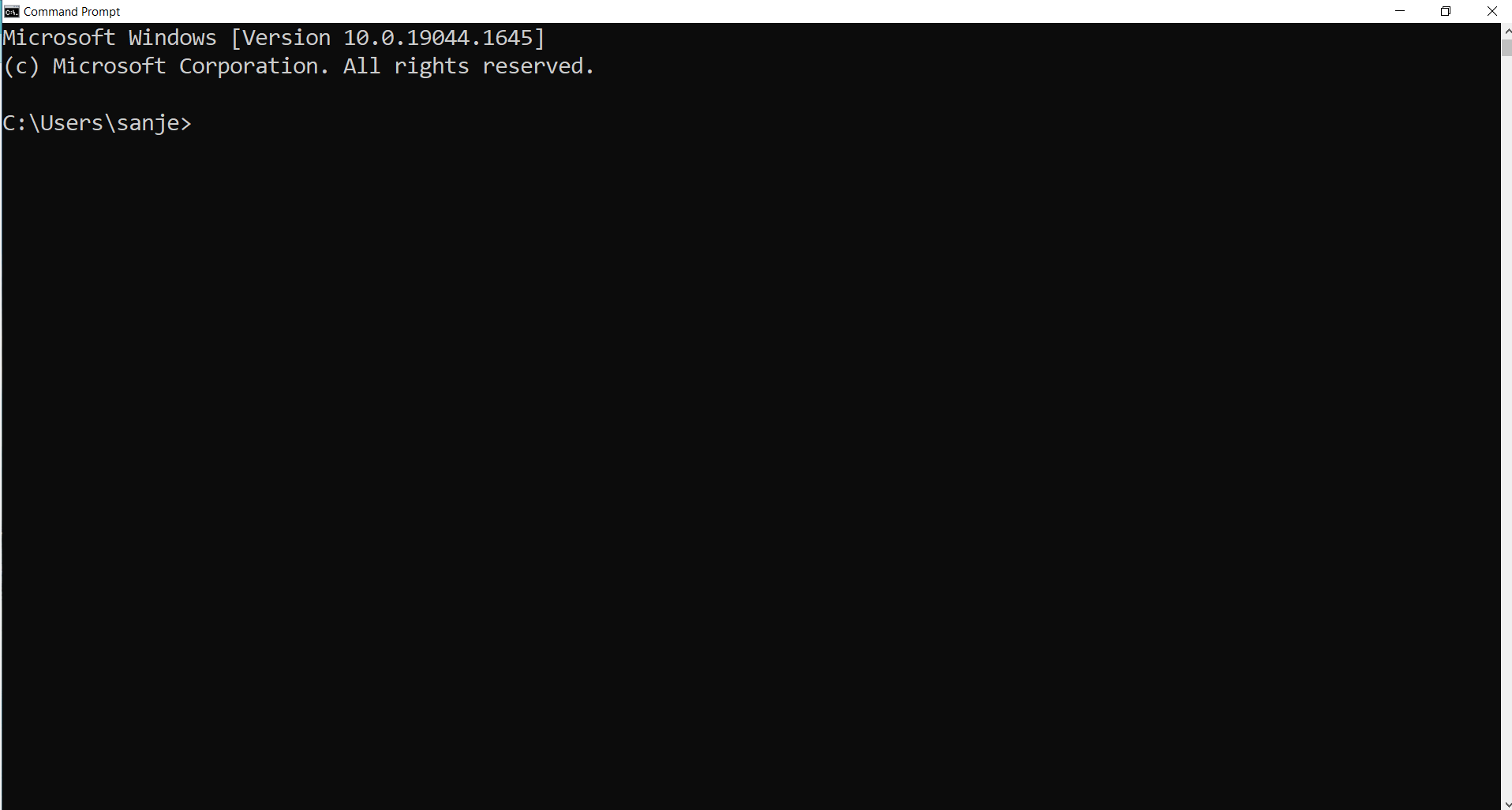
[](https://lucid.app/documents/edit/84c49712-e865-456b-ad48-8e2f5a043c4d/0?callback=close&name=docs&callback_type=back&v=552&s=673)

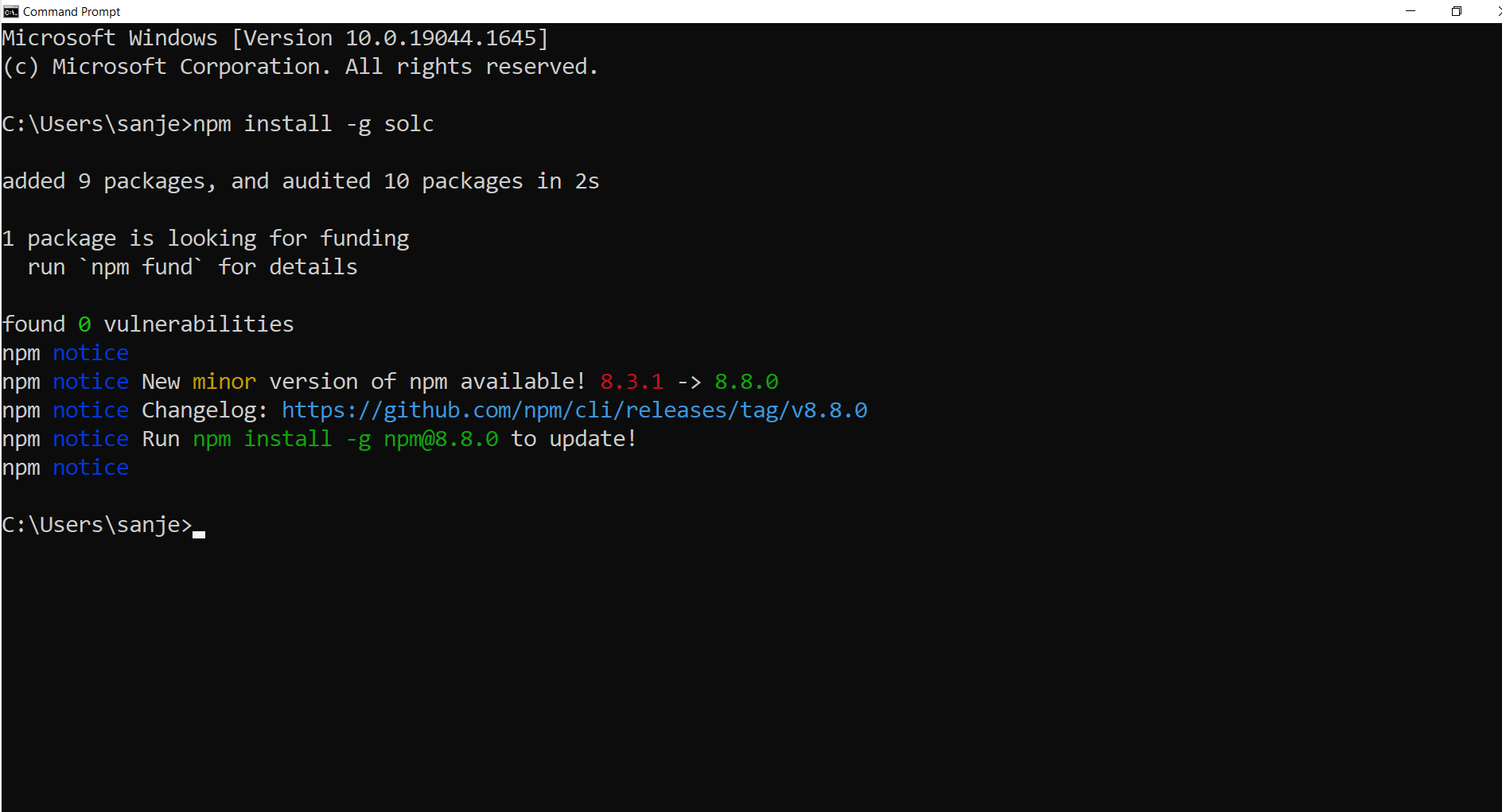
[****](https://lucid.app/documents/edit/91616935-3905-4683-a732-a22ae1d5ddbf/0?callback=close&name=docs&callback_type=back&v=385&s=595.4399999999999)

**Installing Solidity in npm / Node.js**

solcjs is available as a npm package with the name solc. You may install the solcjs npm package locally or globally just like any other npm package. If this package is installed globally, then solcjs, a command-line utility, will be accessible. So, in order to install the command-line utility, perform this command in the command prompt:

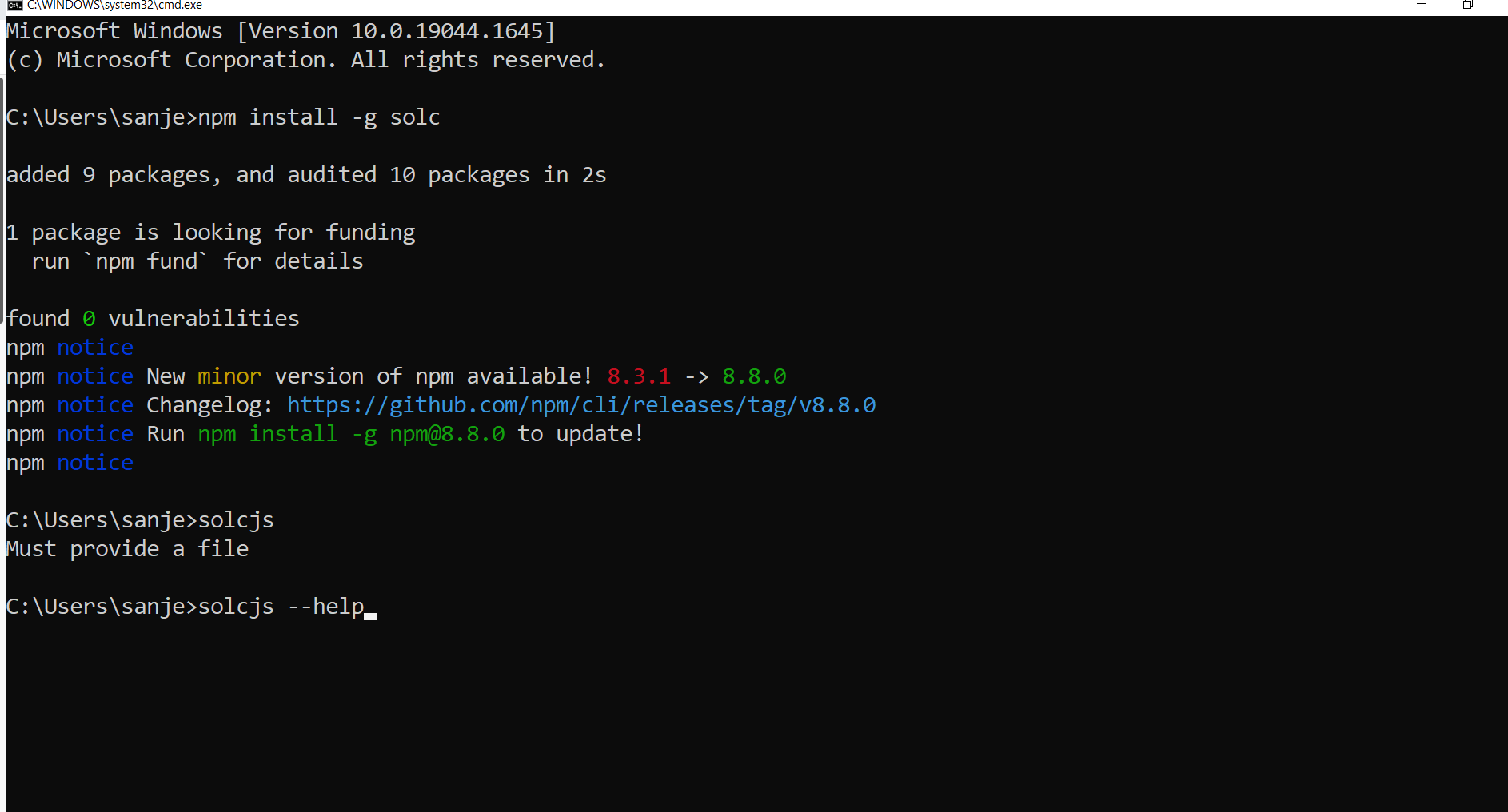
| npm install -g solc |
| --- |

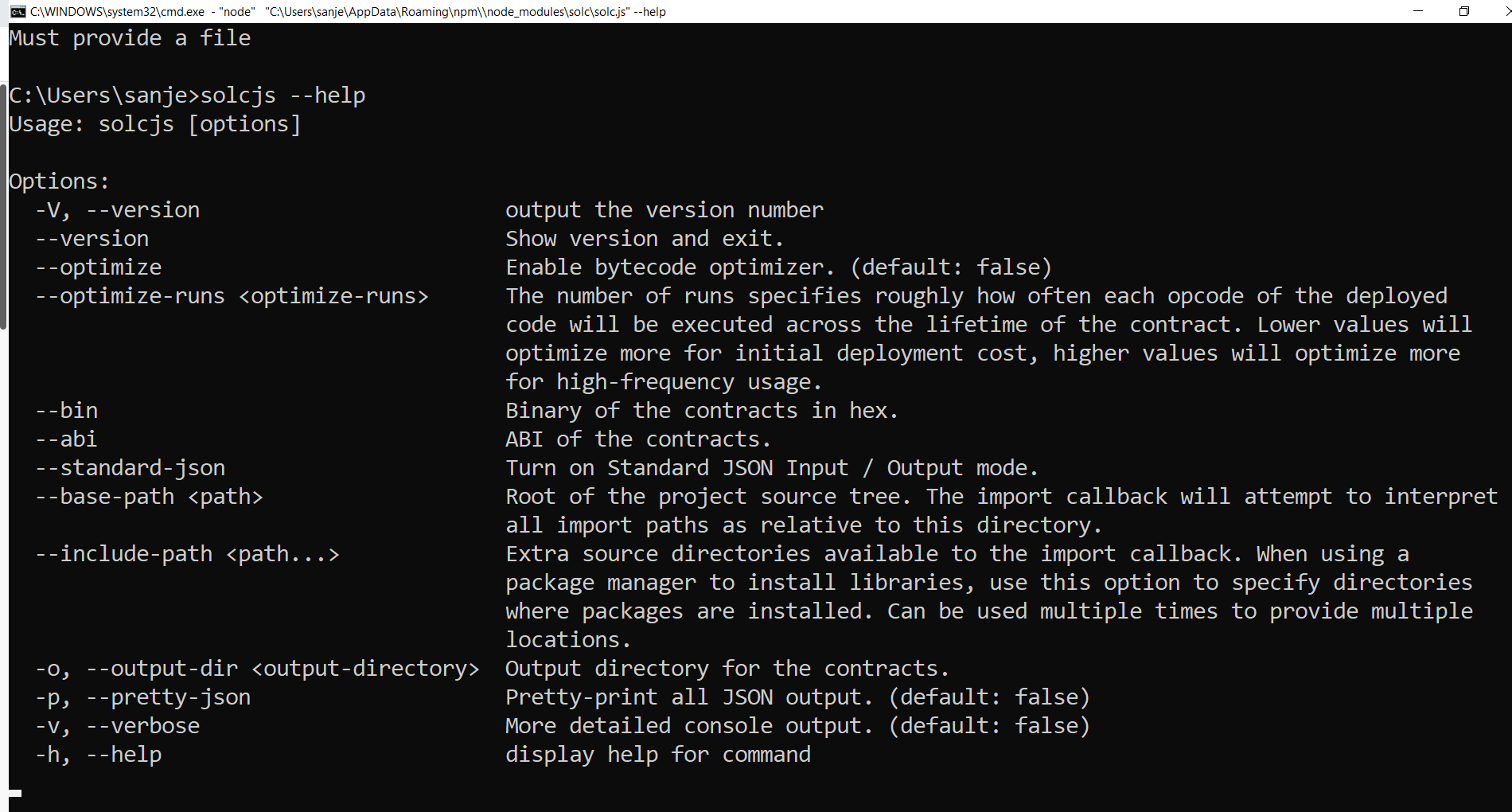




Now go ahead and run this command to see how to compile solidity files using the command-line compiler:

| solcjs —help |
| --- |





To compile a solidity file and get the byte code, go to the directory where you have stored the .sol file and type the following command:

| solcjs —bin <filename>.sol |
| --- |

To get the ABI for the file ,type the following command in the directory where you have stored the .sol file :

| solcjs —abi <filename>.sol |
| --- |

**Layout of a Solidity Source File**

**Solidity Source File Layout-**

The solidity source code can include import directives, pragma directives, struct, enum and function declarations as well as error and constant variable definitions.

**SPDX License Identifier**

The source code of smart contracts may help build trust. The Solidity compiler advocates the use of machine-readable SPDX license IDs since making source code public inevitably raises copyright issues. Every source file should begin with the following license:

| //SPDX-License: MIT |
| --- |

* The compiler does not check the license, but it does include it in the bytecode information.
* If you don't want to indicate a license or the code isn't open source, use UNLICENSED.
* Adding this remark does not relieve you of other licensing duties, such as mentioning a particular license header in each source file or the original copyright holder.
* The compiler recognises the comment anywhere in the file, although it is suggested towards the top.
* The SPDX website has further information on using SPDX license identification.

**Pragmas**

The first line of code in any Solidity file is Pragma.

Pragma indicates the compiler version for the current Solidity file.

Solidity is a new programming language that is always being improved. It releases a new version whenever a new feature or upgrade is added. The current version is 0.8.13 as of the time this document is written. Using the pragma directive, you may choose the compiler version and target your code as follows:

| pragma solidity ^0.8.13; |
| --- |

Although it is not mandatory, it is a good practice to declare the pragma directive as the first statement in a Solidity file. The syntax for the pragma directive is as follows:

| pragma solidity <version number> ; |
| --- |

The version number has two parts: a major build and a minor build.

The previous example's major build number is 8 and minor build number is 13. Minor versions have fewer or no major changes, whereas major versions may have considerable changes. Choose the option that best meets your needs.

The caret ’^’ character is optional in version numbers, although it is important in determining the version number based on the following rules:

The ^ character denotes the most recent major version. So, ^0.8.0 relates to the current build number 8, which is 0.8.13. The character will only target the primary build given.

The Solidity file will only compile with a major build of 8. No other major build will compile it.

It is best to build Solidity programmes using a specific compiler version rather than utilizing ‘^’ . Newer versions may deprecate your code if you use pragma. For example, the toss statement was deprecated in favor of newer constructs like assert, need, and reverse. You don't want to be startled when your code begins acting strange.

**Comments in Solidity**

Solidity, like any other programming language, allows commenting code. Solidity supports three sorts of comments:

1. Single line comments
2. Multiline comments
3. Natural Ethereum Specs (Natspec)

Multiline comments are denoted by /\* and \*/, whereas single-line comments are indicated by //. Natspec offers two formats: /// for single-line comments combination of /\*\* for beginning and \*/ for the end of multiline comments. Natspec is a documentation tool with its own standard.

Let us examine the Solidity comments in the code below:

| // This is a single-line comment in Solidity   /\* This is a multiline comment In Solidity.  Use this when multiple consecutive lines Should be commented as a whole \*/ |
| --- |

**Structure of a Smart Contract**

Solidity contracts are like classes in modern Object Oriented Programming languages. State variables ,Functions, Function Modifiers, Events, Errors, Struct and Enum Types may all be declared in a contract. Contracts may also inherit from other contracts.

Libraries and interfaces are two types of special contracts.

Solidity is used to construct smart contracts for Ethereum. EVMs use smart contracts to deploy and execute their code.

A contract contains multiple constructs:

* **State variables**
* **Structures**
* **Modifiers**
* **Declaratives**
* **Enumerations**
* **Functions**

A typical Smart Contract consists of all the preceding constructs.