The Solidity CheatSheet

Created by **JS Mastery**

Visit ***jsmastery.pro*** for more

Introduction

Solidity is an object-oriented programming language for writing smart contracts. It is used for implementing smart contracts on various blockchain platforms.

Learning Solidity doesn't mean that you are restricted to only the Ethereum Blockchain; it will serve you well on other Blockchains. Solidity is the primary programming language for writing smart contracts for the Ethereum blockchain.

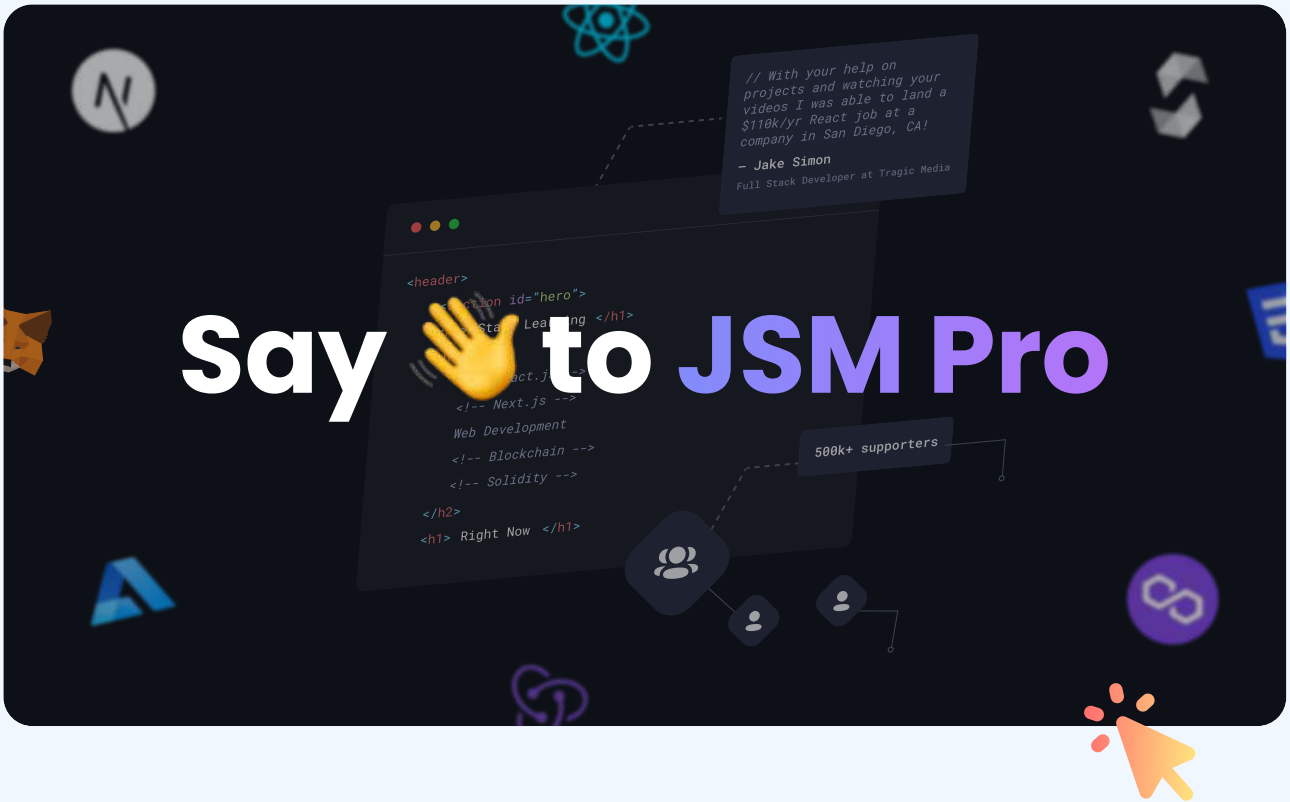
A great way to experiment with Solidity is to use an online IDE called Remix. With Remix, you can load the website, start coding and run your first smart contract.

Don’t worry if you are a beginner and have no idea about how Solidity works, this cheat sheet will give you a quick reference of the keywords, variables, syntax and basics that you must know to get started.

Brought to you by JSM

This guide will provide you with useful information and actionable steps, but if you truly want to dominate the competition and secure a high-paying job as a full-stack software developer, jsmastery.pro is the answer.

Read until the end for more information and **special discounts**! 🙌



Data Types

Data type is a particular kind of data defined by the values it can take

Boolean

Logical:

!	Logical negation
&&	AND
||	OR

Comparisons:

==	equality
!=	inequality

Data Types

Bitwise operators

&	AND
|	OR
^	Bitwise XOR
~	Bitwise negation
<<	Left Shift
>>	Right Shift

Data Types

Arithmetic Operators

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
++	Increment
--	Decrement

Data Types

Relational Operators

`<=`	Less than or equal to
`<`	Less than
`==`	equal to
`!=`	Not equal to
`>=`	Greater than or equal to
`>`	Greater than

Data Types

Assignment Operators

=	Simple Assignment
+=	Add Assignment
-=	Subtract Assignment
*=	Multiply Assignment
/=	Divide Assignment
%=	Modulus Assignment

Value Types

Boolean

This data type accepts only two values.

True or False.

Integer

This data type is used to store integer values, int and uint are used to declare signed and unsigned integers respectively.

Address

Address hold a 20-byte value which represents the size of an Ethereum address. An address can be used to get balance or to transfer a balance by balance and transfer method respectively.

Value Types

Bytes and Strings

Bytes are used to store a fixed-sized character set while the string is used to store the character set equal to or more than a byte. The length of bytes is from 1 to 32, while the string has a dynamic length.

Enums

It is used to create user-defined data types, used to assign a name to an integral constant which makes the contract more readable, maintainable, and less prone to errors. Options of enums can be represented by unsigned integer values starting from 0.

Reference Types

Arrays

An array is a group of variables of the same data type in which variable has a particular location known as an index. By using the index location, the desired variable can be accessed.

Array can be dynamic or fixed size array.

```
uint[] dynamicSizeArray;
```

```
uint[7] fixedSizeArray;
```

Reference Types

Struct

Solidity allows users to create and define their own type in the form of structures. The structure is a group of different types even though it's not possible to contain a member of its own type. The structure is a reference type variable which can contain both value type and reference type

New types can be declared using Struct

```
struct Book {
    string title;
    string author;
    uint book_id;
}
```

Reference Types

Mapping

Mapping is a most used reference type, that stores the data in a key-value pair where a key can be any value types. It is like a hash table or dictionary as in any other programming language, where data can be retrieved by key.

```
mapping(_KeyType => _ValueType)
```

_KeyType can be any built-in types plus bytes and string. No reference type or complex objects are allowed.

_ValueType - can be any type.

Import files

Syntax to import the files

```
import "filename";
```

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

or

```
import "filename" as jsmLogo;
```

```
import {jsmLogo1 as alias, jsmLogo2} from "filename";
```

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

Function Visibility Specifiers

`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

`anonymous`

for events: Does not store event signature as topic

Modifiers

`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

Modifiers

`constant`

for state variables: Disallows assignment (except initialisation), does not occupy storage slot

`immutable`

for state variables: Allows exactly one assignment at construction time and is constant afterwards. Is stored in code

Global Variables

`block.basefee (uint)`

current block's base fee

`block.chainid (uint)`

current chain id

`block.coinbase (address payable)`

current block miner's address

`block.difficulty (uint)`

current block difficulty

`block.gaslimit (uint)`

current block gaslimit

Global Variables

`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)`

sender of the message (current call)

Global Variables

`msg.value (uint)`

number of wei sent with the message

`tx.gasprice (uint)`

gas price of the transaction

`tx.origin (address)`

sender of the transaction (full call chain)

`assert(bool condition)`

abort execution and revert state changes if condition is false (use for internal error)

Global Variables

`require(bool condition)`

abort execution and revert state changes if condition is false

`require(bool condition, string memory message)`

abort execution and revert state changes if condition is false

`revert()`

abort execution and revert state changes

`revert(string memory message)`

abort execution and revert state changes providing an explanatory string

Global Variables

`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

Global Variables

`addmod(uint x, uint y, uint k) returns (uint)`

abort execution and revert state changes if condition is false

`mulmod(uint x, uint y, uint k) returns (uint)`

compute (x * y) % k where the multiplication is performed with arbitrary precision & does not wrap around at 2**256

`this`

(current contract's type): the current contract, explicitly convertible to address or address payable

Global Variables

`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>.code (bytes memory)`

code at the Address (can be empty)

Global Variables

`<address>.codehash (bytes32)`

the codehash of the Address

`<address payable>.send(uint256 amount) returns (bool)`

send given amount of Wei to Address, returns false on failure

`type(C).name (string)`

the name of the contract

`type(C).creationCode (bytes memory)`

creation bytecode of the given contract

Global Variables

`type(C).runtimeCode (bytes memory)`

runtime bytecode of the given contract

`type(I).interfaceId (bytes4)`

value containing the EIP-165 interface identifier of the given interface

`type(T).min (T)`

the minimum value representable by the integer type T

`type(T).max (T)`

the maximum value representable by the integer type T

Global Variables

```
abi.decode(bytes memory encodedData, (...)) returns (...)
```

ABI-decodes the provided data. The types are given in parentheses as second argument

```
abi.encode(...) returns (bytes memory)
```

ABI-encodes the given arguments

```
abi.encodePacked(...) returns (bytes memory)
```

Performs packed encoding of the given arguments.

Global Variables

```
abi.encodeWithSelector(bytes4 selector, ...) returns (bytes memory)
```

ABI-encodes the given arguments starting from the second and prepends the given four-byte selector

```
abi.encodeCall(function functionPointer, (...)) returns (bytes memory)
```

ABI-encodes a call to functionPointer with the arguments found in the tuple. Performs a full type-check, ensuring the types match the function signature

```
abi.encodeWithSignature(string memory signature, ...) returns (bytes memory)
```

Equivalent to

```
abi.encodeWithSelector(bytes4(keccak256(bytes(signature)), ...)
```

Global Variables

```
bytes.concat(...) returns (bytes memory)
```

Concatenates variable number of arguments to one byte array

```
string.concat(...) returns (string memory)
```

Concatenates variable number of arguments to one string array

Reserved Keywords

- `after`
- `alias`
- `apply`
- `auto`
- `byte`
- `case`
- `copyof`
- `default`

Reserved Keywords

- `define`
- `final`
- `implements`
- `in`
- `inline`
- `let`
- `macro`
- `match`

Reserved Keywords

- `mutable`
- `null`
- `of`
- `partial`
- `promise`
- `reference`
- `relocatable`
- `sealed`

Reserved Keywords

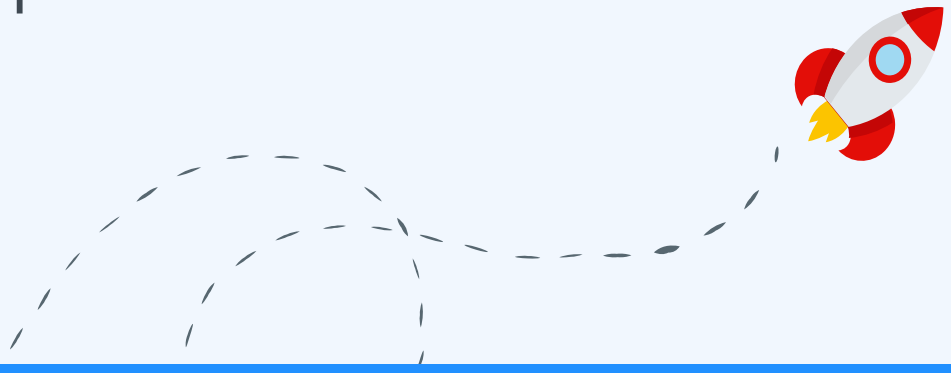
- `sizeof`
- `static`
- `supports`
- `switch`
- `typedef`
- `typeof`
- `var`

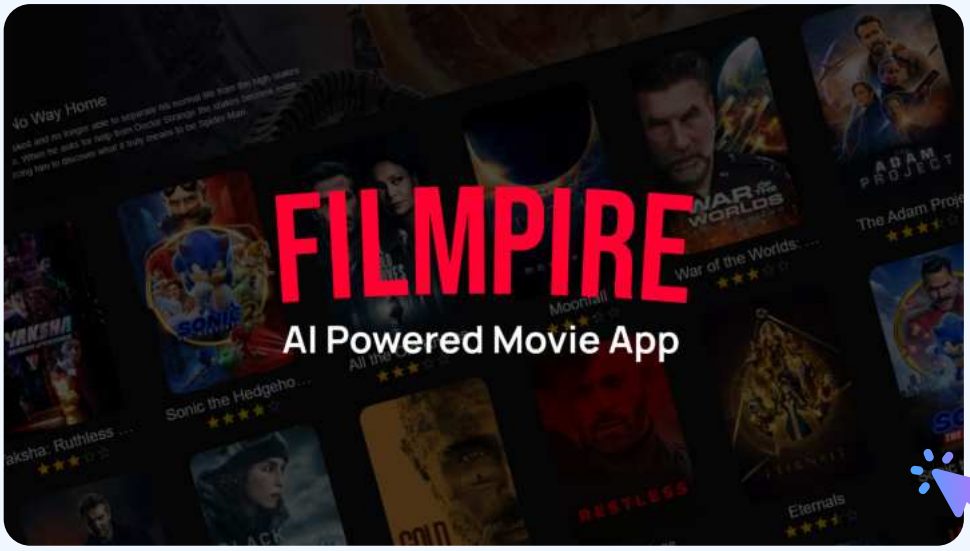
JS Mastery Pro

Looking to advance your career and understand the concepts & technologies that top-shelf employers are looking for?

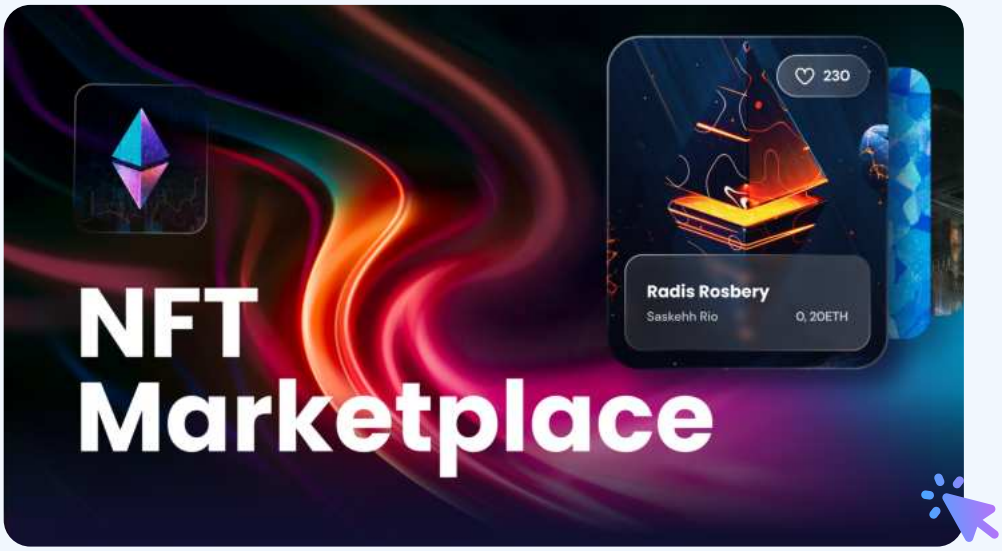
JS Mastery Pro offers two courses that will help you master libraries, tools, and technologies such as React.js, Next.js, Material UI, Solidity, Redux, and many more.

If your goal is to earn a high income while working on projects you love, JS Mastery Pro can help you develop your skills to become a top candidate for lucrative employment and freelance positions.





Become a React.js master as you create a stunning Netflix clone streaming app to showcase movies, actor bios, and more with advanced AI voice functionality.



Leverage Web 3.0 and blockchain technology to build a comprehensive NFT platform where users can discover, create, purchase, & sell non-fungible tokens.

Plus, if you really want to make a splash and add multiple group projects to your portfolio, join the JSM Masterclass Experience to set yourself above the rest and impress hiring managers.

Collaborate with other developers on exciting monthly group projects, have your code reviewed by industry experts, and participate in mock interviews and live Q&As. With two masterclass options available, this is the best way to truly launch your programming career and secure the job of your dreams!

Visit jsmastery.pro today to get started!