Rise in

This cheat sheet is for your future reference. It may seem long but don't worry, you don't need to know these 🙂 You can use this sheet for quick look ups, or simply referencing the syntax for your development. If you want to know more, you can always check the official documentation: Cheatsheet — Solidity 0.8.21 documentation

1. Comments

Comments are used to annotate the code to make it easier to understand. They are ignored by the Solidity compiler.

- **Single-line comments**: Use `//` to comment out a single line.
- **Multi-line comments**: Use `/* */` to comment out multiple lines.

```
// This is a single-line comment
/* This is a multi-line comment
   that spans several lines */
```

2. Pragma Directive

The `pragma` directive helps to lock the Solidity compiler version for your smart contract. This ensures that your contract will only compile with a compatible compiler.

```
pragma solidity ^0.8.0;
```

3. Import

The `import` statement is used to import code from other Solidity files. This is useful for code reusability and separation of concerns.

```
import "./SomeContract.sol";
```

4. Data Types

Solidity supports various data types:

- **uint**: Unsigned integers, cannot be negative.
- **int**: Signed integers, can be negative.
- **bool**: Boolean, true or false.
- **address**: Ethereum address.

- **bytes**: Fixed-size byte arrays.
- **string**: Dynamic-size string.

```
uint256 x = 42;
bool isTrue = true;
address user = msg.sender;
```

5. Enums

Enums help in creating custom types with a range of predefined constants, making code more readable and less error-prone.

```
enum State { Created, Locked, Inactive }
State public state = State.Created;
```

6. Structs

Structs allow you to create complex data types that group variables under a single name.

```
struct Person {
    string name;
    uint age;
}
Person public person = Person("Alice", 30);
```

7. Arrays

Arrays are data structures that can hold more than one value at a time. They can be fixed-size or dynamic.

```
uint[] public numbers = [1, 2, 3];
```

8. Mappings

Mappings are key-value stores that allow you to link one data type to another.

```
mapping(address => uint) public balances;
```

9. Functions

Functions are the building blocks of a Solidity contract. They define executable code and can read and modify contract state.

```
function add(uint x, uint y) public pure returns (uint) {
    return x + y;
```

10. Function Modifiers

Modifiers are reusable pieces of code that can change the behavior of functions. They are often used for access control.

```
modifier onlyOwner() {
    require(msg.sender == owner, "Not the owner");
    _;
}
```

11. Events

Events provide a logging mechanism for the blockchain. They are crucial for frontend applications to "listen" to changes in smart contracts.

```
event LogData(uint indexed data);

emit LogData(42);
```

12. Error Handling

Solidity uses `require`, `assert`, and `revert` for error handling:

- **require**: Used for validating inputs and conditions. Consumes less gas.
- **assert**: Used for internal errors.
- **revert**: Similar to require but allows you to revert complex operations.

```
require(x > 0, "x must be greater than 0");
```

13. Inheritance

Inheritance allows a contract to acquire properties and behavior (i.e., state variables and functions) of a parent contract.

```
contract Child is Parent {
    // code
}
```

14. Interfaces

Interfaces define a contract's external functions and enable the interaction between different contracts.

```
interface IERC20 {
    function transfer(address to, uint256 value) external returns
(bool);
```

15. Libraries

Libraries are reusable pieces of code that can be called from within your contracts. They are deployed only once at a specific address and linked to your contract.

```
using SafeMath for uint;

uint x = y.add(z);
```

16. Storage and Memory

- **Storage**: Persistent data storage that forms the contract state. Changes are very costly in terms of gas.
- **Memory**: Temporary data storage. It's erased between external function calls and is cheaper to use than storage.

```
uint[] storage myArray;
uint[] memory tempArray;
```

17. Payable

The `payable` keyword allows a function to receive Ether.

```
function deposit() public payable {
    // code
}
```

18. Fallback and Receive Functions

These functions are executed when a contract receives Ether without a function being called.

- **Fallback**: A default function marked with `fallback`.
- **Receive**: Explicit function to receive Ether, must be marked `external payable`.

```
fallback() external payable {
    // code
}

receive() external payable {
    // code
}
```

19. Constructor

constructor is a special function that gets executed only once when the contract is ployed.

```
constructor() {
    owner = msg.sender;
}
```

20. Visibility

- **public**: Accessible from this and other contracts.
- **private**: Accessible only from this contract.
- **internal**: Like `private` but also accessible in derived contracts.
- **external**: Only accessible from other contracts.

```
uint public x;
uint private y;
```

21. View and Pure Functions

- **view**: Functions that read the state but don't modify it.
- **pure**: Functions that neither read nor modify the state.

```
function getView() public view returns (uint) {
    return x;
}

function getPure() public pure returns (uint) {
    return 42;
}
```

☆☆☆☆☆ Complete ‹ ›

Rise together in web3

Website

Bootcamps

What is the
Web3 Bootcamp?

Community

Partner with Us

Blog

Events

Bootcamps

Soroban Accelerated Bootcamp in India

Soroban Online Bootcamp

Courses

Rust

Solidity

Solana

Web3 Fundamentals

Blockchain Basics

Company

About Us

Join Us

FAQ

Terms Conditions

Privacy Policy

GDPR Notice

Cookies