Ishanvi
Info-Soft Pvt. Ltd.

solidity.iisplindia.com

Solidity 0.8

YouTube Channel Link:

Solidity Github Source File

{CODE}

Solidity was proposed in August 2014 by Gavin Wood. The language was later developed by the Ethereum project's Solidity team, led by Christian Reitwiessner. Solidity is the primary language on Ethereum as well as on other private blockchains, such as the enterprise-oriented Hyperledger Fabric blockchain.

1. Solidity is an object-oriented, high-level language for implementing smart contracts. Smart contracts are programs that govern the behavior of accounts within the Ethereum state.
2. Solidity is designed to target the Ethereum Virtual Machine (EVM). It is influenced by C++, Python, and JavaScript.
3. Solidity is statically typed, supports inheritance, libraries, and complex user-defined types.
4. With Solidity, you can create contracts for uses such as voting, crowdfunding, blind auctions, and multi-signature wallets.

This Solidity E-Learniong series aimed at helping smart contract developers build better contracts and applications on Ethereum or EVM-based blockchains.

The series covers aspects of the Solidity smart contract language, including addresses, mappings, bytes, structs, arrays, interface, and many more...

Solidity

- Solidity is an object-oriented, high-level language for implementing smart contracts. Smart contracts are programs that govern the behavior of accounts within the Ethereum state.
- Solidity is designed to target the Ethereum Virtual Machine (EVM). It is influenced by C++, Python, and JavaScript.
- Solidity is statically typed, supports inheritance, libraries, and complex user-defined types.
- With Solidity, you can create contracts for uses such as voting, crowdfunding, blind auctions, and multi-signature wallets.

Remix

- Remix IDE is used for the entire journey of smart contract development by users at every knowledge level. It requires no setup, fosters a fast development cycle, and has a rich set of plugins with intuitive GUIs. The IDE comes in two flavors (web app or desktop app) and as a VSCode extension.
- Remix Online IDE, see: https://remix.ethereum.org
- Supported browsers: Firefox, Chrome, Brave.

Bibliography/References

https://docs.soliditylang.org/en/v0.8.20/

https://remix-ide.readthedocs.io/en/latest/#

Pragma

- Pragma is generally the first line of code within any Solidity file. pragma is a directive that specifies the compiler version to be used for current Solidity file.
- Solidity is a new language and is subject to continuous improvement on an on-going basis. Whenever a new feature or improvement is introduced, it comes out with a new version. The current version at the time of writing was 0.8.18.
- With the help of the pragma directive, you can choose the compiler version and target your code accordingly, as shown in the following code example:
- pragma Solidity ^0.8.18;
- Although it is not mandatory, it is a good practice to declare the pragma directive as the first statement in a Solidity file.

SPDX License List

- The SPDX (Software Package Data Exchange) License List is a list of commonly found licenses and exceptions used in free and open source and other collaborative software or documentation.
- The purpose of the SPDX License List is to enable easy and efficient identification of such licenses and exceptions in an SPDX document, in source files or elsewhere.
- The SPDX License List includes a standardized short identifier, full name, vetted license text including matching guidelines markup as appropriate, and a canonical permanent URL for each license and exception.

Contracts

- 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.
- 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.

YouTube Link:

https://www.youtube.com/embed/RyUBESJiV-g

Bibliography/References

1. https://www.oreilly.com
2. https://spdx.org/licenses/preview/#:~:text=The%20SPDX%20License%20List%20is,in%20source%20files%20or%20elsewhere.
3. https://docs.soliditylang.org/en/v0.8.19/contracts.html

Value DataTypes Details

- *Signed/ Unsigned integers* - Integer data types store whole numbers, with signed integers storing both positive and negative values and unsigned integers storing non-negative values.
- *Booleans* - Boolean data type is declared with the bool keyword, and can hold only two possible constant values, true or false.
- *Fixed-point numbers* - Fixed point numbers represent decimal numbers in Solidity, although they aren't fully supported by the Ethereum virtual machine yet.
- *Addresses* - The address type is used to store Ethereum wallet or smart contract addresses, typically around 20 bytes. An address type can be suffixed with the keyword "payable", which restricts it to store only wallet addresses and use the transfer and send crypto functions.
- *Byte arrays* - Byte arrays, declared with the keyword "bytes", is a fixed-size array used to store a predefined number of bytes up to 32, usually declared along with the keyword (bytes1, bytes2).
- *Literals* - Literals are immutable values such as addresses, rationals and integers, strings, unicode and hexadecimals, which can be stored in a variable.
- *Enums* - Enums, short for Enumerable, are a user-defined data type, that restrict the value of the variable to a particular set of constants defined within the program.
- *Contract & Function Types* - Similar to other object oriented languages, contract and function types are used to represent classes and their functions respectively. Contracts contain functions that can modify the contract's state variables.

Value Datatype and Keywords

Type	Keyword	Details
Boolean	bool	true/false
Integer	int/uint	Signed and unsigned integers of varying sizes
Integer	int8 to int256	Signed int from 8 bits to 256 bits. int256 is the same as int.
Integer	uint8 to uint256	Unsigned int from 8 bits to 256 bits. uint256 is the same as uint.
Fixed Point Numbers	fixed/unfixed	Signed and unsigned fixed point numbers of varying sizes.
Addresses	address	The address type is used to store Ethereum wallet or smart contract addresses, typically around 20 bytes

Reference Datatype

Reference type variables store the location of the data. They don't share the data directly. With the help of reference type, two different variables can refer to the same location where any change in one variable can affect the other one.

Type	Details
Arrays	An array is a group of variables of the same data type in which the variable has a particular location known as an index. By using the index location, the desired variable can be accessed. The array size can be fixed or dynamic.
Strings	Strings are like arrays of characters. When we use them, we might

Type	Details
	occupy bigger or shorter storage space.
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 that can contain both value type and reference type
Mapping	Mapping is the most used reference type, that stores the data in a key-value pair where a key can be any value type. It is like a hash table or dictionary as in any other programming language, where data can be retrieved by key.

YouTube Link:

https://www.youtube.com/embed/m7q2I54xgyE

Solidity Source File

State Variables

- Declared at contract level.
- Permanently stored in contract storage.
- Can be set as constants.
- Expensive to use they cost gas.
- Initialised at declaration using a constructor or after contract deployment by calling Setters Functions.

Local Variables

- Declared inside functions.
- If using the memory keyword and are structure they are allocated at run time memory keyword cannot be used at contract level.
- Memory mainly use with address or string variables.

YouTube Link:

https://www.youtube.com/embed/I1x7-j7U9fY

Solidity Source File

Scope of local variables is limited to function in which they are defined but State variables can have three types of scopes.

- **Public** – Public state variables can be accessed internally as well as via messages. For a public state variable, an automatic getter function is generated.
- **Internal** – Internal state variables can be accessed only internally from the current contract or contract deriving from it without using this.
- **Private** – Private state variables can be accessed only internally from the current contract they are defined not in the derived contract from it.

YouTube Link:

https://www.youtube.com/embed/O56DJrZN7RY

Solidity Source File

Solidity supports the following types of operators

- Arithmetic Operators
- Assignment Operators
- Relational Operators
- Logical Operators
- Ternary Operator
- Bitwise Operators

Arithmetic Operators

Arithmetic operators are used to perform arithmetic or mathematical operations

Operation	Operator Symbol
Addition	+
Subtraction	-
Multiplication	*
Division	/
Increment	++ (add 1)
Decrement	— (subtract 1)
Modulus	% (x % y gives the remainder of the integer division of x by y)
Exponent	** (p**q is p to the power of q)

Assignment Operators

Assignment operators shorten the code for the assignment of a value to a variable

Operation	Operator Symbol
Assignment	=
Add assignment	+=
Subtract assignment	-=
Multiply assignment	*=
Divide Assignment	/=
Modulus Assignment	%=

Relational Operators

Relational operators are the ones you will use in your conditions to compare two Data values or results

Operator Symbol	Operator Details
Assignment	=
== (equal)	returns true if two values are equal, false otherwise. IT'S TWO EQUAL SIGNS!!! The single equal sign is the operator used to assign a value to a variable, don't confuse them!
!= (not equal)	returns true if two values are not equal, false otherwise.
> (greater than)	returns true if the left value is greater than the right

Operator Symbol	Operator Details
	value, false otherwise.
>= (greater than or equal to)	returns true if the left value is greater than or equal to the right value, false otherwise.
< (less than)	returns true if the left value is less than the right value, false otherwise.
<= (less than or equal to)	returns true if the left value is less than or equal to the right value, false otherwise.

Logical Operators

Logical operators are used to combine two or more conditions or Boolean results

Operator Symbol	Operator Details
&& (logical AND)	returns true if both conditions are true, returns false otherwise.
|| (logical OR)	returns true if at least one condition is true, returns false otherwise.
! (logical NOT)	returns true if the condition is false and returns false if the condition is true.

Ternary Operators

Ternary operators is a shortcut to deal with simple if / else conditions.
Syntax:

condition ? valueIfConditionIsTrue : valueIfConditionIsFalse

Bitwise Operators

Bitwise operators work directly on bit to perform bit-level operations, with 0 and 1

Operator Symbol	Operator Details
& (bitwise AND)	performs boolean AND operation on each bit of integer argument.
| (bitwise OR)	performs boolean OR operation on each bit of integer argument.
^ (bitwise XOR = Exclusive OR)	performs boolean exclusive OR operation on each bit of integer argument.
~ (bitwise NOT)	performs boolean NOT operation on each bit of integer argument.
<< (left shift)	moves all bits of the first operand to the left by the number of places specified by the second operand.
>> (right shift)	moves all bits of the first operand to the right by the number of places specified by the second operand.

Bibliography/References

1. https://coinsbench.com/solidity-4-operators-conditions-736a2c3faeff

Decision-making statements, also known as conditional statements, determine program direction and flow by specifying boolean expressions evaluated to true or false, enabling or disabling code execution.

- **if statement** – if and else 'if and else' statements check whether a condition is true or false. If the condition is true, it executes some particular set of codes and if the condition is false, it executes some other set of codes.

if statement Syntax

```
if (condition)
{
        Statement or block of code to be executed
        if condition is True
}
```

- **if.. else statement** – if and else 'if and else' statements check whether a condition is true or false. If the condition is true, it executes some particular set of codes and if the condition is false, it executes some other set of codes.

if.. else statement Syntax

```
if (condition)
{
        statement or block of code to be executed
        if condition is True
}
else
{
        statement or block of code to be executed
        if condition is False
}
```

- **if.. else if.. statement** – Chain 'if and else' statements with extra 'else if' for additional conditions.

if.. else if... statement Syntax

```
if (condition1) {
  // code
} else if (condition2) {
  // code
} else if (condition3) {
  // code
} else {
  // code
}
```

YouTube Link:

https://www.youtube.com/embed/YLL1TNrLur8

Solidity Source File

Looping meant, directs a program to perform a set of operations again and again until a specified condition is achieved, which causes the termination of the loop. Programming language Solidity contains three statements for looping:

- while loop
- do… while loop
- for loop

while loop

While loop construct contains the condition first. If the condition is satisfied, the control executes the statements following the while loop else, it ignores these statements. The general form of while loop is:

```
while(condition)
{
statement1;
statement2;
….
}
```

do… while loop

do-while loop construct is another method used in Solidity programming. do-while loop ensures that the program is executed atleast once and checks whether the condition at the end of the do-while loop is true or false. As long as the test condition is true, the statements will be repeated. The control will come out from the loop, only when the test condition is false. The do-while loop has the following form:

```
do
{
statement1;
statement2;
.....
}
while(condition);
```

The blocks of statements with in double braces {} following the word do are executed at least once. Then the condition is evaluated. If the condition is true, the block of statements are executed again until the value of condition tested is false.

for loop

for loop construct is used to execute a set of statements for a given number of times. Thus, it is a shorthand method for executing statements in a loop. The syntax is:

```
for(initial condition; test condition; incrementer or decrementer)
{
 statement1;
 statement2;
}
```

for loop construct requires to specify three characteristics. These are:

1. The initial value of the loop counter;
2. Testing the loop counter value to determine whether its current value has reached the number of repetitions desired;
3. Increasing or decreasing the value of loop counter by a specified number, each time the program segment is executed

YouTube Link:

https://www.youtube.com/embed/rW5KYqWMn7k

Solidity Source File

In Solidity data types are classified into two categories: Value type and Reference type.

1. Strings in Solidity is a reference type of data type which stores the location of the data instead of directly storing the data into the variable.
2. They are dynamic arrays that store a set of characters that can consist of numbers, special characters, spaces, and alphabets.
3. Strings in solidity store data in UTF-8 encoding.
4. Like JavaScript, both Double quote(” “) and Single quote(‘ ‘) can be used to represent strings in solidity.

String and Memory

1. It is important to know that you can not just return a string in solidity because it has to go somewhere, a string has to be stored. In order to store it, we want to let solidity know to store our string to memory.
2. memory is much like RAM.
3. memory in Solidity is a temporary place to store data whereas storage holds data between functions.
4. The Solidity Smart Contract can use any amount of memory during execution, but once the execution stops, the memory is completely wiped off for the next execution.

Escape Characters

Character	Description
\”	Double quote
\’	Single quote
\n	Starts a new line
\\	Backslash
\t	Tab
\r	Carriage return
\b	Backspace
\xNN	Hex escape
\uNNNN	Unicode escape

Bytes to String Conversion

Bytes can be converted to String using string() constructor.

```
bytes memory bstr = new bytes(10);
string message = string(bstr);
```

YouTube Link:

https://www.youtube.com/embed/pcwKAaptTEw

Solidity Source File

Arrays are data structures that store the fixed collection of elements of the same data types in which each and every element has a specific location called index.

Instead of creating numerous individual variables of the same type, we just declare one array of the required size and store the elements in the array and can be accessed using the index.

In Solidity, an array can be of fixed size or dynamic size.

Arrays have a continuous memory location, where the lowest index corresponds to the first element while the highest represents the last.

Fixed-size Arrays

The size of the array should be predefined. The total number of elements should not exceed the size of the array

<data type>[ ] <array name> = <initialization>

Dynamic Arrays

The size of the array is not predefined when it is declared. As the elements are added the size of array changes and at the runtime, the size of the array will be determined.

<data type>[ ] <array name> = <initialization>[Value1,Value2,.....]

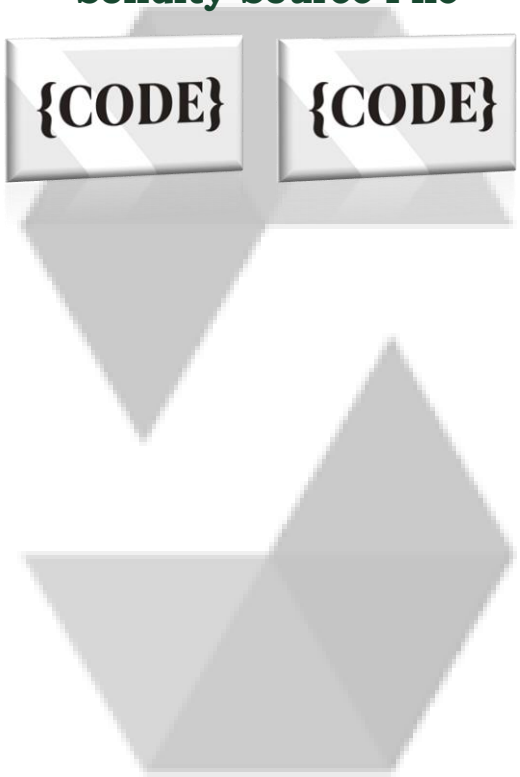
Array Operations

1. Accessing Array Elements
2. Length of Array
3. Push
4. Array slices
5. Pop

YouTube Link:

https://www.youtube.com/embed/CwZJofn1i44

Solidity Source File



Enum(Enumeration)

Enumerations offer an easy way to work with sets of related constants.
An enumeration, or Enum, is a symbolic name for a set of values. Enumerations are treated as data types, and you can use them to create sets of constants for use with variables and properties.

Benefits of using Enumerations

Whenever a procedure accepts a limited set of variables, consider using an enumeration. Enumerations make for clearer and more readable code, particularly when meaningful names are used.

When to Use an Enumeration

1. Reduces errors caused by transposing or mistyping numbers.
2. Makes it easy to change values in the future.
3. Makes code easier to read, which means it is less likely that errors will creep into it.
4. Ensures forward compatibility. With enumerations, your code is less likely to fail if in the future someone changes the values corresponding to the member names.

YouTube Link:

https://www.youtube.com/embed/xyg7Hkis9y0

Solidity Source File

.....

Bibliography/References

https://learn.microsoft.com/en-us/dotnet/visual-basic/programming-guide/language-features/constants-enums/when-to-use-an-enumeration

Mapping in Solidity is a hash table storing key-value pairs, linking unique Ethereum addresses to corresponding value types, similar to dictionaries in other languages.

Any other variable type that accepts a key type and a value type is referred to as mapping.

Syntax

mapping(key => value) <access specifier> <name>;

You can think of mappings as hash tables, Hash tables initialize keys to default values, with zeros representing byte-representations.

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

Please Note:

Mapping can only have type of storage and are generally used for state variables.

Mapping can be marked public. Solidity automatically create getter for it.

YouTube Link:

https://www.youtube.com/embed/x5-vsYxzLG8

Solidity Source File

In Solidity, a struct is a flexible data structure format that allows several data types to be combined into a single variable or a special type. A struct's name designates the subsets of variables it contains after the data types have been grouped into it.

Think of structs as containers that can hold various kinds of objects so that when you move the container, all the contents move with it. As a result, a struct replies in accordance with the data types contained in it when a Solidity developer declares or calls the name of the struct.

1. You can develop more complex data types with various characteristics using Solidity's structures. By building a struct, you can declare whatever type you want.
2. They are helpful for collecting related data into one category.
3. Structures may be imported into one contract from another after being declared outside of it. It often serves as a record representation. The struct keyword, which generates a new data type, is used to define a structure.

Syntax

```
struct <structure_name>
{
   <data type> variable_1;
   <data type> variable_2;
}
```

Declaration

1. Declare a Struct Inside a Contract
2. Declare a Struct outside a Contract with Import Access

YouTube Link:

https://www.youtube.com/embed/1hYj_rrpMas

Solidity Source File

Global Variable List:

Sr.No.	Global Variable	Description
1	blockhash(uint blockNumber) returns (bytes32)	Hash of the given block - only works for 256 most recent, excluding current, blocks
2	block.coinbase (address payable)	Current block miner's address.
3	block.difficulty (uint)	current block difficulty.
4	block.gaslimit (uint)	Current block gaslimit.
5	block.number (uint)	Current block number.
6	block.timestamp	Current block timestamp as seconds since unix epoch.
7	gasleft() returns (uint256)	Remaining gas.
8	**msg.data (bytes calldata)**	**Complete calldata.**
9	**msg.sender (address payable)**	**Sender of the message (current call).**

10	msg.sig (bytes4)	First four bytes of the calldata (i.e. function identifier)
11	**msg.value (uint)**	**Number of wei sent with the message.**
12	now (uint)	Current block timestamp (alias for block.timestamp).
13	tx.gasprice (uint)	Gas price of the transaction.
14	tx.origin (address payable)	Sender of the transaction (full call chain).

Functions are code blocks that perform specific tasks for different input parameters without repeating lines of code. For example, a function can add two numbers by taking the input and summating the numbers without repeating the same lines. This approach simplifies the process of performing tasks without repeating lines of code.

Solidity function visibility specifiers

The visibility of the function decides who can access the function, based on this there are four visibility specifies.

1. **Private**: accessible only inside the current contract.
2. **Internal**: accessible inside the current contract and in child contracts.
3. **External**: can be accessed only outside the contract.
4. **Public**: these are accessible from everywhere, from inside and outside of a contract, also public functions create getter functions for state variables.

Solidity function state mutability

Functions in solidity have certain behavior which can be determined by the state mutability of that function, which tells us how they interact with data stored on the blockchain.

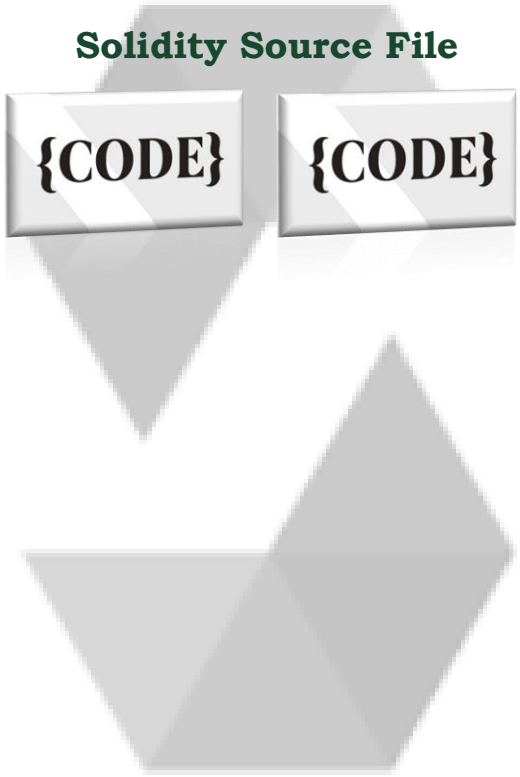
Functions in solidity can be declared as view, pure or payable.

1. **View**: functions that read the state (data stored on blockchain) but do not modify the state are declared as view.
2. **Pure**: these are neither read nor modify the state and are declared as pure functions.
3. **Payable**: functions declared as payable allow the function to send and receive the ether, if a it is not marked as payable it will reject the ether sent to it.

YouTube Link:

https://www.youtube.com/embed/lxcOjaipbkE

Solidity Source File



Solidity's code is encapsulated in contracts which means a contract in Solidity is a collection of code (its functions) and data (its state) that resides at a specific address on the Ethereum blockchain. A contract is a fundamental block of building an application on Ethereum.

POP vs OOP vs BOP

Sr.No.	Title	Short	Language	Details
1	Procedure Oriented Programming	POP	C-Language	Functions and Procedure
2	Object Oriented Programming	OOP	CPP / Java / C# Languages	Classes State & Functions
3	**Block Oriented Programming**	**BOP**	**Solidity Language**	**Contracts State & Functions**

Contract Declaration Flow:

Pragma:

Pragmas are instructions to the compiler on how to treat the code. All solidity source code should start with a "version pragma" which is a declaration of the version of the solidity compiler this code should use. This helps the code from being incompatible with the future versions of the compiler which may bring changes.

Contract:

The contract keyword declares a contract under which the code is encapsulated.

State variables:

State variables are permanently stored in contract storage that is they are written in Ethereum Blockchain.

A function:

Functions are code blocks that perform specific tasks for different input parameters without repeating lines of code. The most common way to define a function in Solidity is by using the function keyword, followed by a unique function name, a list of parameters (that might be empty), and a statement block surrounded by curly braces.

YouTube Link:

https://www.youtube.com/embed/5AmL611Kdl0

Solidity Source File

.....

Bibliography/References

https://spdx.dev/about/

A constructor is a function in Solidity that contains initialization logic for state variables. It is invoked only when the smart contract is deployed, and cannot be invoked again.

If not defined explicitly, the compiler creates a default constructor. To declare a constructor, use the constructor keyword followed by the access specifier, which can be public or internal. A contract with an internal constructor is considered abstract and cannot be deployed.

When inheriting a contract, the child contract must provide the parent contract's constructor parameters. If the child contract doesn't, it's marked as abstract and not deployed. If no explicit constructor is defined, the default constructor will be called.

Key Points:

1. A contract has a single constructor.
2. Executed once to initialize its state.
3. The final code, including public functions and code accessible through public functions, is deployed to the blockchain.
4. Constructors can be public or internal, and if no constructor is defined, a default constructor is present.

YouTube Link:

https://www.youtube.com/embed/dYfuCI7Vao8

Solidity Source File

.....

Bibliography/References

https://cryptomarketpool.com/constructor-in-solidity-smart-contracts/

What is Inheritance?
Solidity's inheritance feature enables programmers to extend contractor attributes and properties to derived contracts, allowing developers to modify them through overriding.

This sets Solidity apart from Java, as it allows multiple inheritances, allowing a derived contract to have multiple parent contracts simultaneously. This allows a single contract to inherit from multiple contracts.

Key Points:

1. A derived contract can access all non-private members including state variables and internal methods. But using this is not allowed.
2. Function overriding is allowed provided function signature remains the same. In case of the difference of output parameters, the compilation will fail.
3. We can call a super contract's function using a super keyword or using a super contract name.
4. In the case of multiple inheritances, function calls using super gives preference to most derived contracts.

*Using the **"is"** Keyword in Solidity*

To create a derived (or inheriting) contract, simply use the is keyword, as demonstrated in the example code below:

```
# A is a derived contract of B
contract A is B{
        //Code
}
```

As mentioned earlier, Solidity allows for multiple inheritances. You can implement multiple inheritances in solidity as shown in this sample code:

```
contract A{
        //Code
}

# single inheritance
contract B is A{
        //Code
}

#multiple inheritance
contract C is A,B {
        //Code
}
```

The implementation above has been carefully chosen to demonstrate a particularly interesting case of multiple inheritances in Solidity. Take note that one of the contracts that C is deriving from is also a derived contract. That is, contract B is also derived from contract A.

This is not an error – Solidity allows this type of multiple inheritances as well, and your code should compile without any errors.

Types of Inheritance in Solidity

1. **Single Inheritance**: In Single or single level inheritance the functions and variables of one base contract are inherited to only one derived contract.
2. **Multi-level Inheritance**: It is very similar to single inheritance, but the difference is that it has levels of the relationship between the parent and the child. The child contract derived from a parent also acts as a parent for the contract which is derived from it.
3. **Hierarchical Inheritance**: In Hierarchical inheritance, a parent contract has more than one child contracts. It is mostly used when a common functionality is to be used in different places.
4. **Multiple Inheritance**: In Multiple Inheritance, a single contract can be inherited from many contracts. A parent contract can have more than one child while a child contract can have more than one parent.

YouTube Link:

https://www.youtube.com/embed/aEL8IpsCB10

Solidity Source File

Single Inheritance

Multi-level Inheritance

Hierarchical Inheritance

Multiple Inheritance

Interfaces are a concept in programming languages that separate the declaration of a function from its actual behavior.

In Solidity, interfaces act as contracts or agreements between the interface and any contract that implements it. By using the interface of a contract, users are bound to use the functions declared in the interface within their contract.

The interface serves as the skeleton of a smart contract, providing a rough idea for other smart contracts to build functionalities and implement them within their contracts.

Why do we use Interfaces?

The interface is a tool used to interact with existing smart contracts on the blockchain, such as the ERC20 token standard contract.

Openzeppelin offers an ERC20 interface for customization, allowing users to create ERC20 tokens according to their specifications.

By creating an interface, users can upgrade contracts based on their features and functionalities, but must customize the skeleton and use the declared functions.

For instance, if users want to inherit functions from another contract without access to its code, they can use an interface to call other contracts.

Interfaces Restrictions

So as I mentioned in the introduction, interfaces in solidity are a more restricted form of abstract contracts. Like in abstract contracts we must have at least one function without its implementation, interface adds some more restrictions to that.

Following are the constraints for the interface to use :

1. You can not declare state variables in the interface.
2. You can not use the constructor inside and interface.
3. You are also not allowed to write modifiers in the interface.
4. You can only declare the functions inside the interface and can not define them inside the interface.
5. All declared functions inside the interface must be external.

YouTube Link:

https://www.youtube.com/embed/V1-tLzM_q3c

Solidity Source File

Events are notifications that provide alerts when important events occur, such as new video uploads.

In Solidity, events work by logging important messages on the Ethereum Virtual Machine (EVM), which can be read from blockchain nodes. These logs can be written to the blockchain, making them crucial for smart contract developers.

Events enable communication between smart contracts and their user interfaces, consuming less gas and being cheaper as they are not accessed by smart contracts.

This makes events an essential aspect of smart contract development, as they enable important alerts to the front end or user interface.

Why Use Events?

1. Events are essential in blockchain application building as they help users and clients stay informed about ongoing processes and transactions.
2. They help users understand the status of transactions, allowing them to wait or cancel transactions based on the results of processing.
3. This is particularly useful when communication with smart contracts is limited.
4. Events send updates on ongoing processes to the user interface, enabling users to stay informed about the transaction's progress and make informed decisions.

How to use events in solidity?

1. **Declaring an Event**: To define an event you start with the keyword "event" followed by the name of the event and parameters wrapped inside ().
   event AccountSwitched(address indexed from, bytes32 indexed to);

2. **Emitting an Event**: You emit an event in the respective function by writing a emit keyword followed by the name of the event and the specified parameters wrapped inside the ().
   emit AccountSwitched(from, to);

3. **Parameters passed to the events**: There are two types of parameters you can pass to an event
   a. **Indexed Parameters** – Indexed parameters are searchable parameters and help to query events. They are also called "topics".
   b. **Non-Indexed Parameters** – Non-Indexed parameters are regular parameters passed to an event that is not searchable and are only used to log the messages to the blockchain.

YouTube Link:

https://www.youtube.com/embed/Q1FHFPsYI-w

Solidity Source File

How does error handling work in Solidity?
Solidity is an object-oriented programming language for implementing smart contracts on blockchains like Ethereum.

It uses state-reverting exceptions to handle errors, undoing changes made to the state in the current call and flagging an error to the caller.

Solidity ensures atomicity as a property by reverting state changes in smart contract calls when errors occur.

Developers can directly interact with other contracts by declaring interfaces.

On the Ethereum blockchain, transactions are atomic, meaning they are either fully complete or have no effect on state and are reverted entirely.

Three main Solidity error handling functions?

assert : Assert is a crucial function in programming to check for code that should never be false, preventing impossible scenarios. If it returns a true value, a terminal bug will be displayed and programs will not execute. Unlike require and revert functions, assert consumes gas supply before reversing the program to its original state. Prior to the Byzantium fork, both functions behaved identically, but compiled to distinct opcodes.

require : The require function is a gate check modifier that checks inputs and conditions before execution. It acts as a gate check, preventing logic from accessing the function and producing errors. Require statements declare prerequisites for running the function, which must be satisfied

before code execution. The function accepts a single argument and returns a boolean value of true or false. If the execution is terminated due to a false condition, unused gas is returned to the caller and the state is reversed to the original state. Customer string messages can also be added.

revert : Revert is a function that does not evaluate conditions or depend on states or statements. It handles error types like require but is more suitable for complex logic gates. When called, unused gas is returned, and the state reverts to its original state. Revert also allows adding custom messages, similar to the require function.

Require vs. Revert vs. Assert

Require:

- Used at the beginning of a function
- Validates against illegal input
- Verifies state conditions prior to execution
- Refunds leftover gas

Revert:

- Identical to require
- Useful for more complex logic flow gates (i.e., complicated if-then blocks)
- Refunds leftover gas

Assert:

- Used at the end of a function
- Validates something that is impossible
- Critical for static code analysis tools
- Does not refund leftover gas

YouTube Link:

https://www.youtube.com/embed/InmVj4G_seY

Solidity Source File

Function modifiers can be used to automatically check a condition before executing a function, serving various use cases.

These modifiers can be executed before or after the function's code, and are necessary when a specific condition is not met. If the given condition is not met, the function will not be executed.

Two variations of a function modifier

1. Function modifier with an argument:

```
modifier  modifier_name(unit arg)
{
   // action to be taken
}
```

2. Function modifier without argument:

```
modifier  modifier_name()
{
   // action to be taken
}
```

What is Merge Wildcard?

The **_;** symbol is known as Merge Wildcard and this is replaced by the function definition during execution.

- In other words, after this wildcard has been used, the control is moved to the location where the appropriate function definition is located.
- This symbol is mandatory for all modifiers.
- The modifier may contain this wildcard anywhere.
- When the wildcard is placed at the end of the modifier, the condition is verified and the appropriate function is executed if it is satisfied.
- When it is placed at the beginning, the appropriate function is executed first followed by the condition verification.

YouTube Link:

https://www.youtube.com/embed/Rek2rwA6V_A

Solidity Source File

The solidity fallback function is executed when no other functions match the function identifier or no data is provided.

It is only assigned to unnamed functions and is executed when a contract receives plain Ether without data.

The fallback function must be marked payable to add Ether to the total balance of the contract. Without such a function, the contract cannot receive Ether through regular transactions and throws an exception

Key Points:

1. Declare with fallback() and have no arguments.
2. If it is not marked payable, the contract will throw an exception if it receives plain ether without data.
3. Can not return anything.
4. Can be defined once per contract.
5. It is also executed if the caller meant to call a function that is not available or receive() does not exist or msg.data is not empty.
6. It is mandatory to mark it external.
7. It is limited to 2300 gas when called by another function by using transfer() or send() method . It is so for as to make this function call as cheap as possible.

fallback()

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.

receive()

This function is called for plain Ether transfers, i.e. for every call with empty calldata.

.....

Bibliography/References

https://www.geeksforgeeks.org/solidity-fall-back-function/

A payable function in Solidity is a function that can receive Ether and respond to an Ether deposit for record-keeping purposes.

All about Payable

- In a smart contract, Payable ensures that money goes into and out of the contract. Solidity functions with a modifier Payable can send and receive Ether transactions, but cannot handle transactions with zero Ether values. If a function does not include the payable keyword, the transaction will be automatically rejected. For example, a receive() function with the payable modifier can receive money in the contract, but a send() function without the payable modifier will reject the transaction.

- Fallback payable functions in Solidity are helpful for ensuring transactions go through if someone sends money to the contract without the payable modifier. It is recommended to use a version of a function with the noname and payable modifiers, with the function name being "noname" instead of "payable." "Payable" is the word that describes the function.

Function declaration with no name:

```
function () public payable {}
```

You can define a payable function using the following syntax:

```
function receive() payable {}
function send() payable {}
```

Solidity supports several methods of transferring ether between the contracts.

1. address.send(amount)
2. address.transfer(amount)
3. address.call.value(amount)()

address.send(amountValue)

Send is the first method for sending ether between contracts. It has a 2300 gas limit for a contract's fallback function, which is not enough for multiple events. If send() fails due to gas shortage, it returns false but does not throw an exception. Therefore, it should be inside the require function. If gas is not paid, transactions cannot be submitted to the blockchain, and changes will be rolled back.

address.transfer(amountValue)

The transfer method has a limit of 2300 gas, but developers proposed adding a.gas() modifier to change the limit. Unlike the send() method, the transfer() method throws an exception when it fails, indicating that the transaction was executed incorrectly.

address.call.value(amountValue)()

The call function is the most personalized method for sending ether, but it will return false if an error occurs. The main difference from previous functions is that the.gas(gasLimit) modifier allows setting the gas limit, especially for complex ether payment functions that require a significant amount of gas.

YouTube Link:

https://www.youtube.com/embed/4Clg68-HiQU?si=w-hDSi_5BhlWXXpl

Solidity Source File

.....

Bibliography/References

https://www.showwcase.com/show/33658/payable-functions-in-solidity