Solidity 201 102 (1/2)

Solidity supports multiple inheritance including polymorphism:

a. 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

b. 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.

c. 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.

Solidity 201 102 (2/2)

d. 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. So 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.

Solidity 201 103 (1/2)

Contract Types:

a. Abstract Contracts: Contracts need to be marked as abstract when at least one of their functions is not implemented. They use the *abstract* keyword.

b. Interfaces: They cannot have any functions implemented. There are further restrictions: 1) They cannot inherit from other contracts, but they can inherit from other interfaces 2) All declared functions must be external 3) They cannot declare a constructor 4) They cannot declare state variables. They use the *interface* keyword.

Solidity 201 103 (2/2)

c. Libraries: They are deployed only once at a specific address and their code is reused using the DELEGATECALL opcode. This means that if library functions are called, their code is executed in the context of the calling contract. They use the *library* keyword.

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.

a. 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.

b. The directive may only be used inside a contract, not inside any of its functions.

Base Class Functions: 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

State Variable Shadowing: This 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.

Function Overriding Changes: The overriding function may only change the visibility of the overridden function from external to public. The mutability may be changed to a more strict one following the order: nonpayable can be overridden by view and pure. view can be overridden by pure. payable is an exception and cannot be changed to any other mutability.

Virtual Functions: Functions without implementation have to be marked virtual outside of interfaces. In interfaces, all functions are automatically considered virtual. Functions with private visibility cannot be virtual.

Public State Variable Override: Public state variables can override external functions if the parameter and return types of the function matches the getter function of the variable. 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

Base Constructors: The constructors of all the base contracts will be called following the linearization rules. If the base constructors have arguments, derived contracts need to specify all of them either in the inheritance list or in the derived constructor.

Name Collision Error: It is an error when any of the following pairs in a contract have the same name due to inheritance: 1) a function and a modifier 2) a function and an event 3) an event and a modifier

Library Restrictions: In comparison to contracts, libraries are restricted in the following ways:

a. they cannot have state variables

b. they cannot inherit nor be inherited

c. they cannot receive Ether

d. they cannot be destroyed

e. it can only access state variables of the calling contract if they are explicitly supplied (it would have no way to name them, otherwise)

f. 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

EVM Storage: Storage is a key-value store that maps 256-bit words to 256-bit words and is accessed with EVM's SSTORE/SLOAD instructions. All locations in storage are initialized as zero.

Storage Layout: State variables of contracts are stored in storage in a compact way such that multiple values sometimes use the same storage slot. Except for dynamically-sized arrays and mappings, data is stored contiguously item after item starting with the first state variable, which is stored in slot 0

Storage Layout Packing: For each state variable, a size in bytes is determined according to its type. Multiple, contiguous items that need less than 32 bytes are packed into a single storage slot if possible, according to the following rules:

a. The first item in a storage slot is stored lower-order aligned

b. Value types use only as many bytes as are necessary to store them

c. If a value type does not fit the remaining part of a storage slot, it is stored in the next storage slot

Storage Layout & Structs/Arrays:

a. Structs and array data always start a new slot and their items are packed tightly according to these rules

b. Items following struct or array data always start a new storage slot

c. The elements of structs and arrays are stored after each other, just as if they were given as individual values.

Storage Layout & Inheritance: 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.

Storage Layout & Types: It might be beneficial to use reduced-size types 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.

a. If you are not reading or writing all the values in a slot at the same time, this can have the opposite effect, though: When one value is written to a multi-value storage slot, the storage slot has to be read first and then combined with the new value such that other data in the same slot is not destroyed.

Storage Layout & Ordering: Ordering of storage variables and struct members affects how 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.

Storage Layout for Mappings & Dynamically-sized Arrays: Due to their unpredictable size, mappings and dynamically-sized array types cannot be stored “in between” the state variables preceding and following them. Instead, they are considered to occupy only 32 bytes with regards to the rules above and the elements they contain are stored starting at a different storage slot that is computed using a Keccak-256 hash.

Storage Layout for Dynamic Arrays: If the storage location of the array ends up being a slot p after applying the storage layout rules, this slot stores the number of elements in the array (byte arrays and strings are an exception). Array data is located starting at keccak256(p) and it is laid out in the same way as statically-sized array data would: One element after the other, potentially sharing storage slots if the elements are not longer than 16 bytes. Dynamic arrays of dynamic arrays apply this rule recursively.

Storage Layout for Mappings: For mappings, the slot stays empty, but it is still needed to ensure that even if there are two mappings next to each other, their content ends up at different storage locations. The value corresponding to a mapping key k is located at keccak256(h(k) . p) where . is concatenation and h is a function that is applied to the key depending on its type: 1) for value types, h pads the value to 32 bytes in the same way as when storing the value in memory. 2) for strings and byte arrays, h computes the keccak256 hash of the unpadded data. If the mapping value is a non-value type, the computed slot marks the start of the data. If the value is of struct type, for example, you have to add an offset corresponding to the struct member to reach the member.

Storage Layout for bytes and string: bytes and string are encoded identically. In general, the encoding is similar to byte1[], in the sense that there is a slot for the array itself and a data area that is computed using a keccak256 hash of that slot's position. However, for short values (shorter than 32 bytes) the array elements are stored together with the length in the same slot.

a. if the data is at most 31 bytes long, the elements are stored in the higher-order bytes (left aligned) and the lowest-order byte stores the value length * 2. For byte arrays that store data which is 32 or more bytes long, the main slot p stores length * 2 + 1 and the data is stored as usual in keccak256(p). 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).

EVM Memory: EVM memory is linear and can be addressed at byte level and accessed with MSTORE/MSTORE8/MLOAD instructions. All locations in memory are initialized as zero.

Memory Layout: Solidity places new memory objects at the free memory pointer and memory is never freed. The free memory pointer points to 0x80 initially.

Reserved Memory: Solidity reserves four 32-byte slots, with specific byte ranges (inclusive of endpoints) being used as follows:

a. 0x00 - 0x3f (64 bytes): scratch space for hashing methods

b. 0x40 - 0x5f (32 bytes): currently allocated memory size (aka. free memory pointer)

c. 0x60 - 0x7f (32 bytes): zero slot (The zero slot is used as initial value for dynamic memory arrays and should never be written to)

Memory Layout & Arrays: Elements in memory arrays in Solidity always occupy multiples of 32 bytes (this is even true for byte[], but not for bytes and string).

a. Multi-dimensional memory arrays are pointers to memory arrays

b. The length of a dynamic array is stored at the first slot of the array and followed by the array elements

Free Memory Pointer: 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. Considering the reserved memory, allocatable memory starts at 0x80, which is the initial value of the free memory pointer.

Zeroed Memory: 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.

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*

Inline Assembly: 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.

a. The language used for inline assembly in Solidity is called Yul

b. An inline assembly block is marked by *assembly { ... }*, where the code inside the curly braces is code in the Yul language

Inline Assembly Access to External Variables, Functions and Libraries:

a. You can access Solidity variables and other identifiers by using their name.

b. Local variables of value type are directly usable in inline assembly

c. Local variables that refer to memory/calldata evaluate to the address of the variable in memory/calldata and not the value itself

d. For local storage variables or state variables, a single Yul identifier is not sufficient, since they do not necessarily occupy a single full storage slot. Therefore, 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. Using x itself will result in an error.

e. Local Solidity variables are available for assignments

f. 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.

g. You can assign to the .slot part of a local storage variable pointer. For these (structs, arrays or mappings), the .offset part is always zero. It is not possible to assign to the .slot or .offset part of a state variable, though

Yul Syntax: Yul parses comments, literals and identifiers in the same way as Solidity. Inside a code block, the following elements can be used:

a. literals, i.e. 0x123, 42 or "abc" (strings up to 32 characters)

b. calls to builtin functions, e.g. add(1, mload(0))

c. variable declarations, e.g. let x := 7, let x := add(y, 3) or let x (initial value of 0 is assigned)

d. identifiers (variables), e.g. add(3, x)

e. assignments, e.g. x := add(y, 3)

f. if statements, e.g. if lt(a, b) { sstore(0, 1) }

g. blocks where local variables are scoped inside, e.g. { let x := 3 { let y := add(x, 1) } }

h. switch statements, e.g. switch mload(0) case 0 { revert() } default { mstore(0, 1) }

i. for loops, e.g. for { let i := 0} lt(i, 10) { i := add(i, 1) } { mstore(i, 7) }

j. function definitions, e.g. function f(a, b) -> c { c := add(a, b) }

Solidity v0.6.0 Breaking Semantic Changes - changes where existing code changes its behaviour without the compiler notifying you about it:

a. The resulting type of an exponentiation is the type of the base. It used to be the smallest type that can hold both the type of the base and the type of the exponent, as with symmetric operations. Additionally, signed types are allowed for the base of the exponentiation.

Solidity v0.6.0 Explicitness Requirements:

a. Functions can now only be overridden when they are either marked with the virtual keyword or defined in an interface. Functions without implementation outside an interface have to be marked virtual. When overriding a function or modifier, the new keyword override must be used. When overriding a function or modifier defined in multiple parallel bases, all bases must be listed in parentheses after the keyword like so: override(Base1, Base2).

b. Member-access to length of arrays is now always read-only, even for storage arrays. It is no longer possible to resize storage arrays by assigning a new value to their length. Use push(), push(value) or pop() instead, or assign a full array, which will of course overwrite the existing content. The reason behind this is to prevent storage collisions of gigantic storage arrays.

c. The new keyword abstract can be used to mark contracts as abstract. It has to be used if a contract does not implement all its functions. Abstract contracts cannot

d. Libraries have to implement all their functions, not only the internal ones.

e. The names of variables declared in inline assembly may no longer end in _slot or _offset.

f. Variable declarations in inline assembly may no longer shadow any declaration outside the inline assembly block. If the name contains a dot, its prefix up to the dot may not conflict with any declaration outside the inline assembly block.

g. State variable shadowing is now disallowed. 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.

Solidity v0.6.0 Semantic and Syntactic Changes:

a. Conversions from external function types to address are now disallowed. Instead external function types have a member called address, similar to the existing selector member.

b. The function push(value) for dynamic storage arrays does not return the new length anymore (it returns nothing).

c. The unnamed function commonly referred to as “fallback function” was split up into a new fallback function that is defined using the fallback keyword and a receive ether function defined using the receive keyword.

d. If present, the receive ether function is called whenever the call data is empty (whether or not ether is received). This function is implicitly payable.

e. The new fallback function is called when no other function matches (if the receive ether function does not exist then this includes calls with empty call data). You can make this function payable or not. If it is not payable then transactions not matching any other function which send value will revert. You should only need to implement the new fallback function if you are following an upgrade or proxy pattern.

Solidity v0.6.0 New Features:

a. The try/catch statement allows you to react on failed external calls.

b. struct and enum types can be declared at file level.

c. Array slices can be used for calldata arrays, for example abi.decode(msg.data[4:], (uint, uint)) is a low-level way to decode the function call payload.

d. Natspec supports multiple return parameters in developer documentation, enforcing the same naming check as @param.

e. Yul and Inline Assembly have a new statement called leave that exits the current function.

f. Conversions from address to address payable are now possible via payable(x), where x must be of type address.

Solidity v0.7.0 Breaking Semantic Changes - changes where existing code changes its behaviour without the compiler notifying you about it:

a. Exponentiation and shifts of literals by non-literals (e.g. 1 << x or 2 ** x) will always use either the type uint256 (for non-negative literals) or int256 (for negative literals) to perform the operation. Previously, the operation was performed in the type of the shift amount / the exponent which can be misleading.

Solidity v0.7.0 Changes to the Syntax - changes that might cause existing contracts to not compile anymore:

a. In external function and contract creation calls, Ether and gas is now specified using a new syntax: x.f{gas: 10000, value: 2 ether}(arg1, arg2). The old syntax – x.f.gas(10000).value(2 ether)(arg1, arg2) – will cause an error.

b. The global variable now is deprecated, block.timestamp should be used instead. The single identifier now is too generic for a global variable and could give the impression that it changes during transaction processing, whereas block.timestamp correctly reflects the fact that it is just a property of the block.

c. NatSpec comments on variables are only allowed for public state variables and not for local or internal variables

d. The token gwei is a keyword now (used to specify, e.g. 2 gwei as a number) and cannot be used as an identifier

e. String literals now can only contain printable ASCII characters and this also includes a variety of escape sequences, such as hexadecimal (\xff) and unicode escapes (\u20ac).

f. Unicode string literals are supported now to accommodate valid UTF-8 sequences. They are identified with the unicode prefix: unicode"Hello 😃 ".

g. State Mutability: The state mutability of functions can now be restricted during inheritance. Functions with default state mutability can be overridden by pure and view functions while view functions can be overridden by pure functions. At the same time, public state variables are considered view and even pure if they are constants.

h. Disallow . in user-defined function and variable names in inline assembly. It is still valid if you use Solidity in Yul-only mode.

i. Slot and offset of storage pointer variable x are accessed via x.slot and x.offset instead of x_slot and x_offset.

Solidity v0.7.0 Removal of Unused or Unsafe Features

a. If a struct or array contains a mapping, it can only be used in storage. Previously, mapping members were silently skipped in memory, which is confusing and error-prone.

b. Assignments to structs or arrays in storage do not work if they contain mappings. Previously, mappings were silently skipped during the copy operation, which is misleading and error-prone.

c. Visibility (public / external) is not needed for constructors anymore: To prevent a contract from being created, it can be marked abstract. This makes the visibility concept for constructors obsolete.

d. Type Checker: Disallow virtual for library functions: Since libraries cannot be inherited from, library functions should not be virtual.

e. Multiple events with the same name and parameter types in the same inheritance hierarchy are disallowed.

f. using A for B only affects the contract it is mentioned in. Previously, the effect was inherited. Now, you have to repeat the using statement in all derived contracts that make use of the feature.

g. Shifts by signed types are disallowed. Previously, shifts by negative amounts were allowed, but reverted at runtime.

h. The finney and szabo denominations are removed. They are rarely used and do not make the actual amount readily visible. Instead, explicit values like 1e20 or the very common gwei can be used.

i. The keyword var cannot be used anymore. Previously, this keyword would parse but result in a type error and a suggestion about which type to use. Now, it results in a parser error.

Solidity v0.8.0 Breaking Semantic Changes - changes where existing code changes its behaviour without the compiler notifying you about it:

a. Arithmetic operations revert on underflow and overflow. You can use unchecked { ... } to use the previous wrapping behaviour. Checks for overflow are very common, so they are the default to increase readability of code, even if it comes at a slight increase of gas costs.

b. ABI coder v2 is activated by default. You can choose to use the old behaviour using *pragma abicoder v1;*. The pragma *pragma experimental ABIEncoderV2;* is still valid, but it is deprecated and has no effect. If you want to be explicit, please use *pragma abicoder v2;* instead.

c. Exponentiation is right associative, i.e., the expression *a**b**c* is parsed as *a**(b**c)*. Before 0.8.0, it was parsed as *(a**b)**c*. This is the common way to parse the exponentiation operator.

d. Failing assertions and other internal checks like division by zero or arithmetic overflow do not use the invalid opcode but instead the revert opcode. More specifically, they will use error data equal to a function call to Panic(uint256) with an error code specific to the circumstances. This will save gas on errors while it still allows static analysis tools to distinguish these situations from a revert on invalid input, like a failing require.

e. If a byte array in storage is accessed whose length is encoded incorrectly, a panic is caused. A contract cannot get into this situation unless inline assembly is used to modify the raw representation of storage byte arrays.

f. If constants are used in array length expressions, previous versions of Solidity would use arbitrary precision in all branches of the evaluation tree. Now, if constant variables are used as intermediate expressions, their values will be properly rounded in the same way as when they are used in run-time expressions.

g. The type byte has been removed. It was an alias of bytes1.

Solidity v0.8.0 New Restrictions - changes that might cause existing contracts to not compile anymore:

a. Explicit conversions from negative literals and literals larger than type(uint160).max to address are disallowed.

b. Explicit conversions between literals and an integer type T are only allowed if the literal lies between type(T).min and type(T).max. In particular, replace usages of uint(-1) with type(uint).max.

c. Explicit conversions between literals and enums are only allowed if the literal can represent a value in the enum.

d. Explicit conversions between literals and address type (e.g. address(literal)) have the type address instead of address payable. One can get a payable address type by using an explicit conversion, i.e., payable(literal).

e. Address literals have the type *address* instead of *address payable*. They can be converted to address payable by using an explicit conversion

f. Function call options can only be given once, i.e. c.f{gas: 10000}{value: 1}() is invalid and has to be changed to c.f{gas: 10000, value: 1}()

g. The global functions log0, log1, log2, log3 and log4 have been removed. These are low-level functions that were largely unused. Their behaviour can be accessed from inline assembly.

h. enum definitions cannot contain more than 256 members. This will make it safe to assume that the underlying type in the ABI is always uint8.

i. Declarations with the name this, super and _ are disallowed, with the exception of public functions and events.

j. The global variables *tx.origin* and *msg.sender* have the type address instead of address payable. One can convert them into address payable by using an explicit conversion.

k. Explicit conversion into address type always returns a non-payable address type

l. The *chainid* builtin in inline assembly is now considered view instead of pure

Zero Address Check: address(0) which is 20-bytes of 0's is treated specially in Solidity contracts because the private key corresponding to this address is unknown. Ether and tokens sent to this address cannot be retrieved and setting access control roles to this address also won't work (no private key to sign transactions). Therefore zero addresses should be used with care and checks should be implemented for user-supplied address parameters.

tx.origin Check: Recall that Ethereum has two types of accounts: Externally Owned Account (EOA) and Contract Account. Transactions can originate only from EOAs. In situations where contracts would like to determine if the *msg.sender* was a contract or not, checking if *msg.sender* is equal to *tx.origin* is an effective check.

Overflow/Underflow Check: Until Solidity version 0.8.0 which introduced checked arithmetic by default, arithmetic was unchecked and therefore susceptible to overflows and underflows which could lead to critical vulnerabilities. The recommended best-practice for such contracts is to use OpenZeppelin's SafeMath library for arithmetic.

OpenZeppelin Libraries: OpenZeppelin's smart contract libraries are perhaps the most commonly used libraries in smart contract projects. These include contracts for popular token standards, access control, security, safe math, proxies and other utilities.

OpenZeppelin ERC20: Implements the popular ERC20 token standard. The functions are:

a. *constructor(string name_, string symbol_)*: Sets the values for name and symbol. The default value of decimals is 18. To select a different value for decimals you should overload it. All three of these values are immutable: they can only be set once during construction.

b. *name() → string*: Returns the name of the token.

c. *symbol() → string*: Returns the symbol of the token, usually a shorter version of the name.

d. *decimals() → uint8*: Returns the number of decimals used to get its user representation. For example, if decimals equals 2, a balance of 505 tokens should be displayed to a user as 5.05 (505 / 10 ** 2). Tokens usually opt for a value of 18, imitating the relationship between Ether and Wei. This is the value ERC20 uses, unless this function is overridden.

e. *totalSupply()*: Returns the amount of tokens in existence.

f. *balanceOf(address account) → uint256*: Returns the amount of tokens owned by account

g. *transfer(address recipient, uint256 amount) → bool*: Moves amount tokens from the caller's account to recipient. Returns a boolean value indicating whether the operation succeeded. Emits a Transfer event.

h. *allowance(address owner, address spender) → uint256*: Returns the remaining number of tokens that spender will be allowed to spend on behalf of owner through transferFrom. This is zero by default. This value changes when approve or transferFrom are called.

i. *approve(address spender, uint256 amount) → bool*: Sets amount as the allowance of spender over the caller's tokens. Returns a boolean value indicating whether the operation succeeded. Emits an Approval event. Warning: changing an allowance with this method brings the risk that someone may use both the old and the new allowance by unfortunate transaction ordering. One possible solution to mitigate this race condition is to first reduce the spender's allowance to 0 and set the desired value afterwards: https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729

j. *transferFrom(address sender, address recipient, uint256 amount) → bool*: Moves amount tokens from sender to recipient using the allowance mechanism. amount is then deducted from the caller's allowance. Returns a boolean value indicating whether the operation succeeded. Emits a Transfer event.

k. *increaseAllowance(address spender, uint256 addedValue) → bool* (Non-standard): Atomically increases the allowance granted to spender by the caller. This is an alternative to approve that can be used as a mitigation for the warning above. Emits an Approval event indicating the updated allowance. Requirement is that spender cannot be the zero address.

l. *decreaseAllowance(address spender, uint256 subtractedValue) → bool* (Non-standard): Atomically decreases the allowance granted to spender by the caller. This is an alternative to approve that can be used as a mitigation for the warning described above. Emits an Approval event indicating the updated allowance. Requirements are: 1) spender cannot be the zero address. 2) spender must have allowance for the caller of at least subtractedValue.

The different extensions/presets are:

1. OpenZeppelin ERC20Burnable: Extension of ERC20 that allows token holders to destroy both their own tokens and those that they have an allowance for, in a way that can be recognized off-chain (via event analysis).
2. OpenZeppelin ERC20Capped: Extension of ERC20 that adds a cap to the supply of tokens and enforces it in the mint function.
3. OpenZeppelin ERC20Pausable: ERC20 token with pausable token transfers, minting and burning. Useful for scenarios such as preventing trades until the end of an evaluation period, or having an emergency switch for freezing all token transfers in the event of a large bug. The *_beforeTokenTransfer()* internal function enforces the not paused condition.
4. OpenZeppelin ERC20PresetFixedSupply: ERC20 token, including: 1) Preminted initial supply 2) Ability for holders to burn (destroy) their tokens 3) No access control mechanism (for minting/pausing) and hence no governance. This contract uses *ERC20Burnable* contract to include burn capabilities

5. OpenZeppelin ERC20Snapshot: This contract extends an ERC20 token with a snapshot mechanism. When a snapshot is created, the balances and total supply at the time are recorded for later access. This can be used to safely create mechanisms based on token balances such as trustless dividends or weighted voting. Snapshots are created by the internal *_snapshot* function, which will emit the Snapshot event and return a snapshot id. To get the total supply at the time of a snapshot, call the function *totalSupplyAt* with the snapshot id. To get the balance of an account at the time of a snapshot, call the *balanceOfAt* function with the snapshot id and the account address.
6. OpenZeppelin ERC20PresetMinterPauser: ERC20 token, including: 1) ability for holders to burn (destroy) their tokens 2) a minter role that allows for token minting (creation) 3) a pauser role that allows to stop all token transfers. This contract uses *AccessControl* contract to lock permissioned functions using the different roles. The account that deploys the contract will be granted the minter and pauser roles, as well as the default admin role, which will let it grant both minter and pauser roles to other accounts.

OpenZeppelin SafeERC20: Wrappers around ERC20 operations that throw on failure when the token contract implementation returns false. Tokens that return no value and instead revert or throw on failure are also supported with non-reverting calls assumed to be successful. Adds *safeTransfer*, *safeTransferFrom*, *safeApprove*, *safeDecreaseAllowance*, and *safeIncreaseAllowance*.

OpenZeppelin TokenTimelock: A token holder contract that will allow a beneficiary to extract the tokens after a given release time. Useful for simple vesting schedules like "advisors get all of their tokens after 1 year".

OpenZeppelin ERC721: Implements the popular ERC721 Non-Fungible Token Standard. The functions are:

a. *balanceOf(address owner) → uint256 balance*: Returns the number of tokens in owner's account.
b. *ownerOf(uint256 tokenId) → address owner*: Returns the owner of the tokenId token. Requirements: tokenId must exist.
c. *transferFrom(address from, address to, uint256 tokenId)*: Transfers tokenId token from from to to. Requirements: from cannot be the zero address. to cannot be the zero address. tokenId token must be owned by from. If the caller is not from, it must be approved to move this token by either approve or setApprovalForAll. Emits a Transfer event.
d. *approve(address to, uint256 tokenId)*: Gives permission to to to transfer tokenId token to another account. The approval is cleared when the token is transferred. Only a single account can be approved at a time, so approving the zero address clears previous approvals. Requirements: 1) The caller must own the token or be an approved operator 2) tokenId must exist. Emits an Approval event.
e. *getApproved(uint256 tokenId) → address operator*: Returns the account approved for tokenId token. Requirements: tokenId must exist.
f. *isApprovedForAll(address owner, address operator) → bool*: Returns if the operator is allowed to manage all of the assets of owner.

g. *safeTransferFrom(address from, address to, uint256 tokenId)*: Safely transfers tokenId token from from to to, checking first that contract recipients are aware of the ERC721 protocol to prevent tokens from being forever locked. Requirements: 1) from cannot be the zero address 2) to cannot be the zero address. 3) tokenId token must exist and be owned by from 4) If the caller is not from, it must be have been allowed to move this token by either approve or setApprovalForAll 5) If to refers to a smart contract, it must implement IERC721Receiver.onERC721Received, which is called upon a safe transfer. Emits a Transfer event. (The use of this function is encouraged over the related but unsafe transferFrom function.)

h. *setApprovalForAll(address operator, bool _approved)*: Approve or remove operator as an operator for the caller. Operators can call transferFrom or safeTransferFrom for any token owned by the caller. Requirements: The operator cannot be the caller. Emits an ApprovalForAll event.

The different extensions/presets/utilities are:

1. OpenZeppelin ERC721Burnable: ERC721 Token that can be irreversibly burned (destroyed).
2. OpenZeppelin ERC721Enumerable: This implements an optional extension of ERC721 defined in the EIP that adds enumerability of all the token ids in the contract as well as all token ids owned by each account.
3. OpenZeppelin ERC721Pausable: ERC721 token with pausable token transfers, minting and burning. Useful for scenarios such as preventing trades until the end of an evaluation period, or having an emergency switch for freezing all token transfers in the event of a large bug.
4. OpenZeppelin ERC721URIStorage: ERC721 token with storage based token URI management.
5. OpenZeppelin ERC721Holder: Implementation of the IERC721Receiver interface. Accepts all token transfers.

6. OpenZeppelin ERC721PresetMinterPauserAutoId: ERC721 token, including: 1) ability for holders to burn (destroy) their tokens 2) a minter role that allows for token minting (creation) 3) a pauser role that allows to stop all token transfers 4) token ID and URI autogeneration. This contract uses AccessControl to lock permissioned functions using the different roles. The account that deploys the contract will be granted the minter and pauser roles, as well as the default admin role, which will let it grant both minter and pauser roles to other accounts.

OpenZeppelin ERC777: Like ERC20, ERC777 is a standard for fungible tokens with improvements such as getting rid of the confusion around decimals, minting and burning with proper events, among others, but its killer feature is receive hooks. ERC777 is backwards compatible with ERC20 (See here)

a. A hook is simply a function in a contract that is called when tokens are sent to it, meaning accounts and contracts can react to receiving tokens. This enables a lot of interesting use cases, including atomic purchases using tokens (no need to do approve and transferFrom in two separate transactions), rejecting reception of tokens (by reverting on the hook call), redirecting the received tokens to other addresses, among many others.

b. Both contracts and regular addresses can control and reject which token they send by registering a tokensToSend hook. (Rejection is done by reverting in the hook function.)

c. Both contracts and regular addresses can control and reject which token they receive by registering a tokensReceived hook. (Rejection is done by reverting in the hook function.)

d. The tokensReceived hook allows to send tokens to a contract and notify it in a single transaction, unlike ERC-20 which requires a double call (approve/transferFrom) to achieve this.

e. Furthermore, since contracts are required to implement these hooks in order to receive tokens, no tokens can get stuck in a contract that is unaware of the ERC777 protocol, as has happened countless times when using ERC20s.

f. It mandates that decimals always returns a fixed value of 18, so there's no need to set it ourselves

g. Has a concept of *defaultOperators* which are special accounts (usually other smart contracts) that will be able to transfer tokens on behalf of their holders

h. Implements *send* (besides *transfer*) where if the recipient contract has not registered itself as aware of the ERC777 protocol then transfers to it are disabled to prevent tokens from being locked forever. Accounts can be notified of tokens being sent to them by having a contract implement this *IERC777Recipient* interface and registering it on the ERC1820 global registry.

OpenZeppelin ERC1155: is a novel token standard that aims to take the best from previous standards to create a fungibility-agnostic and gas-efficient token contract.

a. The distinctive feature of ERC1155 is that it uses a single smart contract to represent multiple tokens at once

b. Accounts have a distinct balance for each token id, and non-fungible tokens are implemented by simply minting a single one of them.

c. This approach leads to massive gas savings for projects that require multiple tokens. Instead of deploying a new contract for each token type, a single ERC1155 token contract can hold the entire system state, reducing deployment costs and complexity.

d. Because all state is held in a single contract, it is possible to operate over multiple tokens in a single transaction very efficiently. The standard provides two functions, *balanceOfBatch* and *safeBatchTransferFrom*, that make querying multiple balances and transferring multiple tokens simpler and less gas-intensive.

OpenZeppelin Ownable: provides a basic access control mechanism, where there is an account (an owner) that can be granted exclusive access to specific functions. By default, the owner account will be the one that deploys the contract. This can later be changed with *transferOwnership*. This module is used through inheritance. It will make available the modifier *onlyOwner*, which can be applied to your functions to restrict their use to the owner.

OpenZeppelin AccessControl: provides a general role based access control mechanism. Multiple hierarchical roles can be created and assigned each to multiple accounts. Roles can be used to represent a set of permissions. *hasRole* is used to restrict access to a function call. Roles can be granted and revoked dynamically via the *grantRole* and *revokeRole* functions which can only be called by the role's associated admin accounts.

While the simplicity of *Ownable* can be useful for simple systems or quick prototyping, different levels of authorization are often needed. You may want for an account to have permission to ban users from a system, but not create new tokens. Role-Based Access Control (RBAC) offers flexibility in this regard. We will effectively be defining multiple roles, each allowed to perform different sets of actions. An account may have, for example, 'moderator', 'minter' or 'admin' roles, which you will then check for instead of simply using *onlyOwner*. Separately, you will be able to define rules for how accounts can be granted a role, have it revoked, and more.

OpenZeppelin AccessControlEnumerable: Extension of AccessControl that allows enumerating the members of each role.

OpenZeppelin Pausable: provides an emergency stop mechanism using functions pause and unpause that can be triggered by an authorized account. This module is used through inheritance. It will make available the modifiers *whenNotPaused* and *whenPaused*, which can be applied to the functions of your contract. Only the functions using the modifiers will be affected when the contract is paused or unpaused.

OpenZeppelin ReentrancyGuard: prevents reentrant calls to a function. Inheriting from ReentrancyGuard will make the *nonReentrant* modifier available, which can be applied to functions to make sure there are no nested (reentrant) calls to them.

OpenZeppelin PullPayment: provides a pull-payment strategy, where the paying contract doesn't invoke any functions on the receiver account which must withdraw its payments itself. Pull-payments are often considered the best practice when it comes to sending Ether, security-wise. It prevents recipients from blocking execution and eliminates reentrancy concerns.

OpenZeppelin Address: Collection of functions related to the address type:

a. *isContract(address account) → bool*: Returns true if account is a contract. It is unsafe to assume that an address for which this function returns false is an externally-owned account (EOA) and not a contract. Among others, isContract will return false for the following types of addresses: 1) an externally-owned account 2) a contract in construction 3) an address where a contract will be created 4) an address where a contract lived, but was destroyed

b. *sendValue(address payable recipient, uint256 amount)*: Replacement for Solidity's transfer: sends amount wei to recipient, forwarding all available gas and reverting on errors. EIP1884 increases the gas cost of certain opcodes, possibly making contracts go over the 2300 gas limit imposed by transfer, making them unable to receive funds via transfer. sendValue removes this limitation.

c. *functionCallWithValue(address target, bytes data, uint256 value) → bytes*: Same as functionCall, but also transferring value wei to target. Requirements: 1) the calling contract must have an ETH balance of at least value. 2) the called Solidity function must be payable.

d. *functionStaticCall(address target, bytes data) → bytes*: Same as functionCall, but performing a static call.

e. *functionCall(address target, bytes data) → bytes*: Performs a Solidity function call using a low level call. A plain `call` is an unsafe replacement for a function call: use this function instead. If target reverts with a revert reason, it is bubbled up by this function (like regular Solidity function calls). Returns the raw returned data. Requirements: target must be a contract. calling target with data must not revert.

f. *functionDelegateCall(address target, bytes data) → bytes*: Same as functionCall, but performing a delegate call.

The above functionCall* functions have variants which pass an *errorMessage* parameter that specifies the fallback revert reason when target reverts.

OpenZeppelin Arrays: Collection of functions related to array types:

a. *findUpperBound(uint256[] array, uint256 element) → uint256*: Searches a sorted array and returns the first index that contains a value greater or equal to element. If no such index exists (i.e. all values in the array are strictly less than element), the array length is returned. Time complexity O(log n). array is expected to be sorted in ascending order, and to contain no repeated elements.

OpenZeppelin Context: Provides information about the current execution context, including the sender of the transaction and its data. While these are generally available via *msg.sender* and *msg.data*, they should not be accessed in such a direct manner, since when dealing with meta-transactions the account sending and paying for execution may not be the actual sender (as far as an application is concerned). This contract is only required for intermediate, library-like contracts.

OpenZeppelin Counters: Provides counters that can only be incremented or decremented by one. This can be used e.g. to track the number of elements in a mapping, issuing ERC721 ids, or counting request ids. Functions are:

a. current(struct Counters.Counter counter) → uint256

b. increment(struct Counters.Counter counter)

c. decrement(struct Counters.Counter counter)

OpenZeppelin Create2: makes usage of the CREATE2 EVM opcode easier and safer. CREATE2 can be used to compute in advance the address where a smart contract will be deployed, which allows for interesting new mechanisms known as 'counterfactual interactions'.

a. *deploy(uint256 amount, bytes32 salt, bytes bytecode) → address*: Deploys a contract using CREATE2. The address where the contract will be deployed can be known in advance via computeAddress.The bytecode for a contract can be obtained from Solidity with type(contractName).creationCode. Requirements: 1) bytecode must not be empty. 2) salt must have not been used for bytecode already. 3) the factory must have a balance of at least amount. 4) if amount is non-zero, bytecode must have a payable constructor.

b. *computeAddress(bytes32 salt, bytes32 bytecodeHash) → address*: Returns the address where a contract will be stored if deployed via deploy. Any change in the bytecodeHash or salt will result in a new destination address.

c. *computeAddress(bytes32 salt, bytes32 bytecodeHash, address deployer) → address*: Returns the address where a contract will be stored if deployed via deploy from a contract located at deployer. If the deployer is this contract's address, it returns the same value as computeAddress.

OpenZeppelin Multicall: Provides a function to batch together multiple calls in a single external call

a. *multicall(bytes[] calldata data) external → bytes[]*: Receives and executes a batch of function calls on this contract

OpenZeppelin Strings: String operations:

a. *toString(uint256 value) → string*: Converts a uint256 to its ASCII string decimal representation.

b. *toHexString(uint256 value) → string*: Converts a uint256 to its ASCII string hexadecimal representation.

c. *toHexString(uint256 value, uint256 length)* → string: Converts a uint256 to its ASCII string hexadecimal representation with fixed length.

OpenZeppelin ECDSA: provides functions for recovering and managing Ethereum account ECDSA signatures. These are often generated via *web3.eth.sign*, and are a 65 byte array (of type bytes in Solidity) arranged the following way: [[v (1)], [r (32)], [s (32)]]. The data signer can be recovered with *ECDSA.recover*, and its address compared to verify the signature. Most wallets will hash the data to sign and add the prefix '\x19Ethereum Signed Message:\n', so when attempting to recover the signer of an Ethereum signed message hash, you'll want to use *toEthSignedMessageHash.*

a. The `ecrecover` EVM opcode allows for malleable (non-unique) signatures. *This library prevents that by requiring the `s` value to be in the lower half order, and the `v` value to be either 27 or 28.*

OpenZeppelin MerkleProof: This deals with verification of Merkle Trees proofs.

a. *verify:* which can prove that some value is part of a Merkle tree. Returns true if a `leaf` can be proved to be a part of a Merkle tree defined by `root`. For this, a `proof` must be provided, containing sibling hashes on the branch from the leaf to the root of the tree. Each pair of leaves and each pair of pre-images are assumed to be sorted.

OpenZeppelin SignatureChecker: Provide a single mechanism to verify both private-key (EOA) ECDSA signature and ERC1271 contract signatures. Using this instead of ECDSA.recover in your contract will make them compatible with smart contract wallets such as Argent and Gnosis.

a. Note: unlike ECDSA signatures, contract signature's are revocable, and the outcome of this function can thus change through time. It could return true at block N and false at block N+1 (or the opposite).

b. Externally Owned Accounts (EOA) can sign messages with their associated private keys, but currently contracts cannot. This is a problem for many applications that implement signature based off-chain methods, since contracts can't easily interact with them as they do not possess a private key. ERC 1271 proposes a standard way for any contracts to verify whether a signature on behalf of a given contract is valid.

OpenZeppelin EIP712: EIP 712 is a standard for hashing and signing of typed structured data. This contract implements the EIP 712 domain separator (*_domainSeparatorV4*) that is used as part of the encoding scheme, and the final step of the encoding to obtain the message digest that is then signed via ECDSA (*_hashTypedDataV4*). Protocols need to implement the type-specific encoding they need in their contracts using a combination of abi.encode and keccak256.

a. *constructor(string name, string version)*: Initializes the domain separator and parameter caches. The meaning of name and version is specified in EIP 712: 1) name is the user readable name of the signing domain, i.e. the name of the DApp or the protocol 2) version: the current major version of the signing domain.

b. *_domainSeparatorV4() → bytes32*: Returns the domain separator for the current chain.

i. *_hashTypedDataV4(bytes32 structHash) → bytes32*: Given an already hashed struct, this function returns the hash of the fully encoded EIP712 message for this domain. This hash can be used together with ECDSA.recover to obtain the signer of a message.

OpenZeppelin Escrow: holds funds designated for a payee until they withdraw them. The contract that uses this escrow as its payment method should be its owner, and provide public methods redirecting to the escrow's deposit and withdraw if the escrow rules are satisfied.

a. *depositsOf(address payee)* → *uint256*:

b. *deposit(address payee)*: Stores the sent amount as credit to be withdrawn.

c. *withdraw(address payable payee)*: Withdraw accumulated balance for a payee, forwarding all gas to the recipient.

OpenZeppelin ConditionalEscrow: Derived from Escrow and only allows withdrawal if a condition is met by providing the *withdrawalAllowed()* function which returns whether an address is allowed to withdraw their funds and is to be implemented by derived contracts.

OpenZeppelin RefundEscrow: Derived from ConditionalEscrow and holds funds for a beneficiary, deposited from multiple parties. The owner account (that is, the contract that instantiates this contract) may deposit, close the deposit period, and allow for either withdrawal by the beneficiary, or refunds to the depositors.

OpenZeppelin ERC165: In Solidity, it's frequently helpful to know whether or not a contract supports an interface you'd like to use. ERC165 is a standard that helps do runtime interface detection using a lookup table. You can register interfaces using *_registerInterface(bytes4)* and *supportsInterface(bytes4 interfaceId)* returns a bool indicating if that interface is supported or not.

OpenZeppelin Math: Standard math utilities missing in the Solidity language:

a. *max(uint256 a, uint256 b)*: Returns the larger of two numbers

b. *min(uint256 a, uint256 b)*: Returns the smaller of two numbers

c. *average(uint256 a, uint256 b)*: Returns the average of two numbers. The result is rounded towards zero.

OpenZeppelin SafeMath: provides mathematical functions that protect your contract from overflows and underflows. Include the contract with using SafeMath for uint256; and then call the functions:

a. myNumber.add(otherNumber): Returns the addition of two unsigned integers, reverting on overflow. Counterpart to Solidity's `+` operator.

b. myNumber.sub(otherNumber): Returns the subtraction of two unsigned integers, reverting on overflow (when the result is negative). Counterpart to Solidity's `-` operator.

c. myNumber.div(otherNumber): Returns the division of two unsigned integers, reverting on overflow. The result is rounded towards zero. Counterpart to Solidity's `/` operator.

d. myNumber.mul(otherNumber): Returns the multiplication of two unsigned integers, reverting on overflow. Counterpart to Solidity's `*` operator.

e. myNumber.mod(otherNumber): Returns the modulus of two unsigned integers, reverting when dividing by zero. Counterpart to Solidity's `%` operator.

The corresponding try* functions return results with an overflow flag instead of reverting.

OpenZeppelin SignedSafeMath: provides the same mathematical functions as SafeMath but for signed integers

a. myNumber.add(otherNumber): Returns the addition of two signed integers, reverting on overflow. Counterpart to Solidity's `+` operator.

b. myNumber.sub(otherNumber): Returns the subtraction of two signed integers, reverting on overflow (when the result is negative). Counterpart to Solidity's `-` operator.

c. myNumber.div(otherNumber): Returns the division of two signed integers, reverting on overflow. The result is rounded towards zero. Counterpart to Solidity's `/` operator.

d. myNumber.mul(otherNumber): Returns the multiplication of two signed integers, reverting on overflow. Counterpart to Solidity's `*` operator.

OpenZeppelin SafeCast: Wrappers over Solidity's uintXX/intXX casting operators with added overflow checks. Downcasting from uint256/int256 in Solidity does not revert on overflow. This can easily result in undesired exploitation or bugs, since developers usually assume that overflows raise errors. `SafeCast` restores this intuition by reverting the transaction when such an operation overflows.

a. *toUint128(uint256 value) returns (uint128)*: Returns the downcasted uint128 from uint256, reverting on overflow (when the input is greater than largest uint128). Similar functions are available for toUint64(uint256 value), toUint32(uint256 value), toUint16(uint256 value), toUint8(uint256 value)

b. *function toInt256(uint256 value) returns (int256)*: Converts an unsigned uint256 into a signed int256

c. *toInt128(int256 value) internal pure returns (uint256)*: Returns the downcasted int128 from int256, reverting on overflow (when the input is less than smallest int128 or greater than largest int128). Similar functions are available for toInt64(int256 value), toInt32(int256 value), toInt16(int256 value), toInt8(int256 value).

d. *function toUint256(int256 value)* returns (uint256): Converts a signed int256 into an unsigned uint256

e. Similar functions downcasting to 224/96/64/32/16/8 bits for both unsigned and signed.

OpenZeppelin EnumerableMap: Library for managing an enumerable variant of Solidity's mapping type. Maps have the following properties: 1) Entries are added, removed, and checked for existence in constant time (O(1)) 2) Entries are enumerated in O(n). No guarantees are made on the ordering. As of v3.0.0, only maps of type uint256 → address (UintToAddressMap) are supported.

a. *set(struct EnumerableMap.UintToAddressMap map, uint256 key, address value) → bool*: Adds a key-value pair to a map, or updates the value for an existing key. Returns true if the key was added to the map, that is if it was not already present.

b. *remove(struct EnumerableMap.UintToAddressMap map, uint256 key) → bool*: Removes a value from a set. Returns true if the key was removed from the map, that is if it was present.

c. *contains(struct EnumerableMap.UintToAddressMap map, uint256 key) → bool*: Returns true if the key is in the map.

d. *length(struct EnumerableMap.UintToAddressMap map) → uint256*: Returns the number of elements in the map.

e. *at(struct EnumerableMap.UintToAddressMap map, uint256 index) → uint256, address*: Returns the element stored at position index in the set. Note that there are no guarantees on the ordering of values inside the array, and it may change when more values are added or removed. Requirements: index must be strictly less than length.

f. *tryGet(struct EnumerableMap.UintToAddressMap map, uint256 key) → bool, address*: Tries to return the value associated with key. Does not revert if key is not in the map.

g. *get(struct EnumerableMap.UintToAddressMap map, uint256 key) → address:* Returns the value associated with key. Requirements: key must be in the map.

OpenZeppelin EnumerableSet: Library for managing sets of primitive types. Sets have the following properties: 1) Elements are added, removed, and checked for existence in constant time (O(1)) 2) Elements are enumerated in O(n). No guarantees are made on the ordering. As of v3.3.0, sets of type bytes32 (Bytes32Set), address (AddressSet) and uint256 (UintSet) are supported.

a. *add(struct EnumerableSet.Bytes32Set set, bytes32 value) → bool*: Add a value to a set. Returns true if the value was added to the set, that is if it was not already present.

b. *remove(struct EnumerableSet.Bytes32Set set, bytes32 value) → bool*: Removes a value from a set. Returns true if the value was removed from the set, that is if it was present.

c. *contains(struct EnumerableSet.Bytes32Set set, bytes32 value) → bool*: Returns true if the value is in the set.

d. *length(struct EnumerableSet.Bytes32Set set) → uint256*: Returns the number of values in the set.

e. *at(struct EnumerableSet.Bytes32Set set, uint256 index) → bytes32*: Returns the value stored at position index in the set. Note that there are no guarantees on the ordering of values inside the array, and it may change when more values are added or removed. Requirements: index must be strictly less than length.

f. *add(struct EnumerableSet.AddressSet set, address value) → bool*: Add a value to a set. Returns true if the value was added to the set, that is if it was not already present.

g. *remove(struct EnumerableSet.AddressSet set, address value) → bool*: Removes a value from a set. Returns true if the value was removed from the set, that is if it was present.

h. *contains(struct EnumerableSet.AddressSet set, address value) → bool*: Returns true if the value is in the set. length(struct EnumerableSet.AddressSet set) → uint256: Returns the number of values in the set.

i. *at(struct EnumerableSet.AddressSet set, uint256 index) → address*: Returns the value stored at position index in the set. Note that there are no guarantees on the ordering of values inside the array, and it may change when more values are added or removed. Requirements: index must be strictly less than length.

j. *add(struct EnumerableSet.UintSet set, uint256 value) → bool*: Add a value to a set. Returns true if the value was added to the set, that is if it was not already present.

k. *remove(struct EnumerableSet.UintSet set, uint256 value) → bool*: Removes a value from a set. Returns true if the value was removed from the set, that is if it was present.

l. *contains(struct EnumerableSet.UintSet set, uint256 value) → bool*: Returns true if the value is in the set. O(1).

m. *length(struct EnumerableSet.UintSet set) → uint256*: Returns the number of values on the set.

n. *at(struct EnumerableSet.UintSet set, uint256 index) → uint256*: Returns the value stored at position index in the set. Note that there are no guarantees on the ordering of values inside the array, and it may change when more values are added or removed. Requirements: index must be strictly less than length.

OpenZeppelin BitMaps: Library for managing uint256 to bool mapping in a compact and efficient way, providing the keys are sequential.

a. struct BitMap: mapping(uint256 => uint256) _data;

b. get(BitMap storage bitmap, uint256 index) → *bool:* Returns whether the bit at `index` is set.

c. setTo(BitMap storage bitmap, uint256 index, bool value): Sets the bit at `index` to the boolean `value`

d. function set(BitMap storage bitmap, uint256 index): Sets the bit at `index`

e. function unset(BitMap storage bitmap, uint256 index): Unsets the bit at `index`

OpenZeppelin PaymentSplitter: allows to split Ether payments among a group of accounts. The sender does not need to be aware that the Ether will be split in this way, since it is handled transparently by the contract. The split can be in equal parts or in any other arbitrary proportion. The way this is specified is by assigning each account to a number of shares. Of all the Ether that this contract receives, each account will then be able to claim an amount proportional to the percentage of total shares they were assigned.

a. PaymentSplitter follows a pull payment model. This means that payments are not automatically forwarded to the accounts but kept in this contract, and the actual transfer is triggered as a separate step by calling the release function.

OpenZeppelin TimelockController: acts as a timelocked controller. When set as the owner of an Ownable smart contract, it enforces a timelock on all *onlyOwner* maintenance operations. This gives time for users of the controlled contract to exit before a potentially dangerous maintenance operation is applied. By default, this contract is self administered, meaning administration tasks have to go through the timelock process. The proposer (resp executor) role is in charge of proposing (resp executing) operations. A common use case is to position this TimelockController as the owner of a smart contract, with a multisig or a DAO as the sole proposer.

a. *constructor(uint256 minDelay, address[] proposers, address[] executors)*: Initializes the contract with a given minDelay.

b. *receive()*: Contract might receive/hold ETH as part of the maintenance process.

c. *isOperation(bytes32 id) → bool pending*: Returns whether an id corresponds to a registered operation. This includes both Pending, Ready and Done operations.

d. *isOperationPending(bytes32 id) → bool pending*: Returns whether an operation is pending or not.

e. *isOperationReady(bytes32 id) → bool ready*: Returns whether an operation is ready or not.

f. *isOperationDone(bytes32 id) → bool done*: Returns whether an operation is done or not.

g. *getTimestamp(bytes32 id) → uint256 timestamp*: Returns the timestamp at which an operation becomes ready (0 for unset operations, 1 for done operations).

h. *getMinDelay() → uint256 duration*: Returns the minimum delay for an operation to become valid. This value can be changed by executing an operation that calls updateDelay.

i. *hashOperation(address target, uint256 value, bytes data, bytes32 predecessor, bytes32 salt) → bytes32 hash*: Returns the identifier of an operation containing a single transaction.

j. *hashOperationBatch(address[] targets, uint256[] values, bytes[] datas, bytes32 predecessor, bytes32 salt) → bytes32 hash*: Returns the identifier of an operation containing a batch of transactions.

k. *schedule(address target, uint256 value, bytes data, bytes32 predecessor, bytes32 salt, uint256 delay)*: Schedule an operation containing a single transaction. Emits a CallScheduled event. Requirements: the caller must have the 'proposer' role.

l. *scheduleBatch(address[] targets, uint256[] values, bytes[] datas, bytes32 predecessor, bytes32 salt, uint256 delay)*: Schedule an operation containing a batch of transactions. Emits one CallScheduled event per transaction in the batch. Requirements: the caller must have the 'proposer' role.

m. *cancel(bytes32 id)*: Cancel an operation. Requirements: the caller must have the 'proposer' role.

n. *execute(address target, uint256 value, bytes data, bytes32 predecessor, bytes32 salt)*: Execute an (ready) operation containing a single transaction. Emits a CallExecuted event. Requirements: the caller must have the 'executor' role.

o. *executeBatch(address[] targets, uint256[] values, bytes[] datas, bytes32 predecessor, bytes32 salt)*: Execute an (ready) operation containing a batch of transactions. Emits one CallExecuted event per transaction in the batch. Requirements: the caller must have the 'executor' role.

p. *updateDelay(uint256 newDelay)*: Changes the minimum timelock duration for future operations. Emits a MinDelayChange event. Requirements: the caller must be the timelock itself. This can only be achieved by scheduling and later executing an operation where the timelock is the target and the data is the ABI-encoded call to this function.

Solidity 201 183

OpenZeppelin ERC2771Context: A Context variant for ERC2771. ERC2771 provides support for meta transactions, which are transactions that have been:

a. Authorized by the Transaction Signer. For example, signed by an externally owned account

b. Relayed by an untrusted third party that pays for the gas (the Gas Relay)

Solidity 201 183+ (1/2)

The problem is that for a contract that is not natively aware of meta transactions, the *msg.sender* of the transaction will make it appear to be coming from the Gas Relay and not the Transaction Signer. A secure protocol for a contract to accept meta transactions needs to prevent the Gas Relay from forging, modifying or duplicating requests by the Transaction Signer. The entities are:

Solidity 201 183+ (2/2)

1. Transaction Signer - entity that signs & sends to request to Gas Relay

2. Gas Relay - receives a signed request off-chain from Transaction Signer and pays gas to turn it into a valid transaction that goes through Trusted Forwarder

3. Trusted Forwarder - a contract that is trusted by the Recipient to correctly verify the signature and nonce before forwarding the request from Transaction Signer

4. Recipient - a contract that can securely accept meta-transactions through a Trusted Forwarder by being compliant with this standard.

Solidity 201 184

OpenZeppelin MinimalForwarder: provides a simple minimal forwarder (as described above) to be used together with an ERC2771 compatible contract. It verifies the nonce and signature of the forwarded request before calling the destination contract.

a. struct ForwardRequest {address from; address to; uint256 value; uint256 gas; uint256 nonce; bytes data;}

b. verify(ForwardRequest calldata req, bytes calldata signature) public view → (bool)

c. execute(ForwardRequest calldata req, bytes calldata signature) → (success, returndata)

OpenZeppelin Proxy: This abstract contract provides a fallback function that delegates all calls to another contract using the EVM instruction delegatecall. We refer to the second contract as the implementation behind the proxy, and it has to be specified by overriding the virtual _implementation function. Additionally, delegation to the implementation can be triggered manually through the _fallback function, or to a different contract through the _delegate function. The success and return data of the delegated call will be returned back to the caller of the proxy.

a. *_delegate(address implementation)*: Delegates the current call to implementation. This function does not return to its internal call site, it will return directly to the external caller.

b. *_implementation() → address*: This is a virtual function that should be overridden so it returns the address to which the fallback function and *_fallback* should delegate.

c. *_fallback()*: Delegates the current call to the address returned by *_implementation()*. This function does not return to its internal call site, it will return directly to the external caller.

d. *fallback()*: Fallback function that delegates calls to the address returned by *_implementation()*. Will run if no other function in the contract matches the call data.

e. *receive()*: Fallback function that delegates calls to the address returned by *_implementation()*. Will run if call data is empty.

f. *_beforeFallback()*: Hook that is called before falling back to the implementation. Can happen as part of a manual _fallback call, or as part of the Solidity fallback or receive functions. If overridden, should call *super._beforeFallback()*.

OpenZeppelin ERC1967Proxy: implements an upgradeable proxy. It is upgradeable because calls are delegated to an implementation address that can be changed. This address is stored in storage in the location specified by EIP1967, so that it doesn't conflict with the storage layout of the implementation behind the proxy. Upgradeability is only provided internally through *_upgradeTo*.

a. *constructor(address _logic, bytes _data)*: Initializes the upgradeable proxy with an initial implementation specified by _logic. If _data is nonempty, it's used as data in a delegate call to _logic. This will typically be an encoded function call, and allows initializing the storage of the proxy like a Solidity constructor.

b. *_implementation() → address impl*: Returns the current implementation address.

c. *_upgradeTo(address newImplementation)*: Upgrades the proxy to a new implementation. Emits an Upgraded event.

OpenZeppelin TransparentUpgradeableProxy: implements a proxy that is upgradeable by an admin. To avoid proxy selector clashing, which can potentially be used in an attack, this contract uses the transparent proxy pattern. This pattern implies two things that go hand in hand: 1) If any account other than the admin calls the proxy, the call will be forwarded to the implementation, even if that call matches one of the admin functions exposed by the proxy itself 2) If the admin calls the proxy, it can access the admin functions, but its calls will never be forwarded to the implementation. If the admin tries to call a function on the implementation it will fail with an error that says "admin cannot fallback to proxy target".

These properties mean that the admin account can only be used for admin actions like upgrading the proxy or changing the admin, so it's best if it's a dedicated account that is not used for anything else. This will avoid headaches due to sudden errors when trying to call a function from the proxy implementation.

1. *constructor(address _logic, address admin_, bytes _data)*: Initializes an upgradeable proxy managed by _admin, backed by the implementation at _logic, and optionally initialized with _data.
2. *admin() → address admin_*: Returns the current admin.
3. *implementation() → address implementation_*: Returns the current implementation.
4. *changeAdmin(address newAdmin)*: Changes the admin of the proxy. Emits an AdminChanged event.
5. *_admin() → address adm*: Returns the current admin.

6. *upgradeTo(address newImplementation)*: Upgrade the implementation of the proxy.
7. *upgradeToAndCall(address newImplementation, bytes data)*: Upgrade the implementation of the proxy, and then call a function from the new implementation as specified by data, which should be an encoded function call. This is useful to initialize new storage variables in the proxied contract.
8. *_beforeFallback()*: Makes sure the admin cannot access the fallback function.

OpenZeppelin ProxyAdmin: This is an auxiliary contract meant to be assigned as the admin of a TransparentUpgradeableProxy.

a. *getProxyImplementation(contract TransparentUpgradeableProxy proxy) → address*: Returns the current implementation of proxy. Requirements: This contract must be the admin of proxy.

b. *getProxyAdmin(contract TransparentUpgradeableProxy proxy) → address*: Returns the current admin of proxy. Requirements: This contract must be the admin of proxy.

c. *changeProxyAdmin(contract TransparentUpgradeableProxy proxy, address newAdmin)*: Changes the admin of proxy to newAdmin. Requirements: This contract must be the current admin of proxy.

d. *upgrade(contract TransparentUpgradeableProxy proxy, address implementation)*: Upgrades proxy to implementation. Requirements: This contract must be the admin of proxy.

e. *upgradeAndCall(contract TransparentUpgradeableProxy proxy, address implementation, bytes data)*: Upgrades proxy to implementation and calls a function on the new implementation. Requirements: This contract must be the admin of proxy.

OpenZeppelin BeaconProxy: implements a proxy that gets the implementation address for each call from a UpgradeableBeacon. The beacon address is stored in storage slot uint256(keccak256('eip1967.proxy.beacon')) - 1, so that it doesn't conflict with the storage layout of the implementation behind the proxy.

a. *constructor(address beacon, bytes data)*: Initializes the proxy with beacon. If data is nonempty, it's used as data in a delegate call to the implementation returned by the beacon. This will typically be an encoded function call, and allows initializing the storage of the proxy like a Solidity constructor. Requirements: beacon must be a contract with the interface IBeacon.

b. *_beacon() → address beacon*: Returns the current beacon address.

c. *_implementation() → address*: Returns the current implementation address of the associated beacon.

d. *_setBeacon(address beacon, bytes data)*: Changes the proxy to use a new beacon. If data is nonempty, it's used as data in a delegate call to the implementation returned by the beacon. Requirements: 1) beacon must be a contract 2) The implementation returned by beacon must be a contract.

OpenZeppelin UpgradeableBeacon: is used in conjunction with one or more instances of BeaconProxy to determine their implementation contract, which is where they will delegate all function calls. An owner is able to change the implementation the beacon points to, thus upgrading the proxies that use this beacon.

a. *constructor(address implementation_)*: Sets the address of the initial implementation, and the deployer account as the owner who can upgrade the beacon.

b. *implementation() → address*: Returns the current implementation address.

c. *upgradeTo(address newImplementation)*: Upgrades the beacon to a new implementation. Emits an Upgraded event. Requirements: 1) msg.sender must be the owner of the contract 2) newImplementation must be a contract.

OpenZeppelin Clones: EIP 1167 is a standard for deploying minimal proxy contracts, also known as "clones". To simply and cheaply clone contract functionality in an immutable way, this standard specifies a minimal bytecode implementation that delegates all calls to a known, fixed address. The library includes functions to deploy a proxy using either create (traditional deployment) or create2 (salted deterministic deployment). It also includes functions to predict the addresses of clones deployed using the deterministic method.

a. *clone(address implementation) → address instance*: Deploys and returns the address of a clone that mimics the behaviour of implementation. This function uses the create opcode, which should never revert.

b. *predictDeterministicAddress(address implementation, bytes32 salt) → address predicted*: Computes the address of a clone deployed using Clones.cloneDeterministic.

c. *cloneDeterministic(address implementation, bytes32 salt) → address instance*: Deploys and returns the address of a clone that mimics the behaviour of implementation. This function uses the create2 opcode and a salt to deterministically deploy the clone. Using the same implementation and salt multiple times will revert, since the clones cannot be deployed twice at the same address.

d. *predictDeterministicAddress(address implementation, bytes32 salt, address deployer)* → address predicted: Computes the address of a clone deployed using Clones.cloneDeterministic.

OpenZeppelin Initializable: aids in writing upgradeable contracts, or any kind of contract that will be deployed behind a proxy. Since a proxied contract cannot have a constructor, it is common to move constructor logic to an external initializer function, usually called *initialize*. It then becomes necessary to protect this initializer function so it can only be called once. The initializer modifier provided by this contract will have this effect.

To avoid leaving the proxy in an uninitialized state, the initializer function should be called as early as possible by providing the encoded function call as the _data argument. When used with inheritance, manual care must be taken to not invoke a parent initializer twice, or to ensure that all initializers are idempotent. This is not verified automatically as constructors are by Solidity.

Dappsys DSProxy: implements a proxy deployed as a standalone contract which can then be used by the owner to execute code. A user would pass in the bytecode for the contract as well as the calldata for the function they want to execute.The proxy will create a contract using the bytecode. It will then delegatecall the function and arguments specified in the calldata.

Dappsys DSMath: provides arithmetic functions for the common numerical primitive types of Solidity. You can safely add, subtract, multiply, and divide uint numbers without fear of integer overflow. You can also find the minimum and maximum of two numbers. Additionally, this package provides arithmetic functions for two new higher level numerical concepts called *wad* (18 decimals) and *ray* (27 decimals). These are used to represent fixed-point decimal numbers. A wad is a decimal number with 18 digits of precision and a ray is a decimal number with 27 digits of precision. These functions are necessary to account for the difference between how integer arithmetic behaves normally, and how decimal arithmetic should actually work.

The standard functions are the uint set, so their function names are not prefixed: *add*, *sub*, *mul*, *min*, and *max*. There is no div function, as divide-by-zero checking is built into the Solidity compiler. The int functions have an *i* prefix: *imin* and *imax*. Wad functions have a *w* prefix: *wmul*, *wdiv*. Ray functions have a *r* prefix: *rmul*, *rdiv*, *rpow*.

Dappsys DSAuth: Provides a flexible and updatable auth pattern which is completely separate from application logic. By default, the auth modifier will restrict function-call access to the including contract owner and the including contract itself. The auth modifier provided by DSAuth triggers the internal isAuthorized function to require that the msg.sender is authorized ie. the sender is either: 1) the contract owner 2) the contract itself or 3) has been granted permission via a specified authority.

Dappsys DSGuard: Manages an Access Control List which maps source and destination addresses to function signatures. Intended to be used as an authority for DSAuth where it acts as a lookup table for the *canCall* function to provide boolean answers as to whether a particular address is authorized to call a given function at another address. The ACL is a mapping of *[src][dst][sig] => boolean* where an address src can be either permitted or forbidden access to a function *sig* at address *dst* according to the boolean value. When used as an authority by DSAuth the *src* is considered to be the *msg.sender*, the *dst* is the including contract and *sig* is the function which invoked the auth modifier.

Dappsys DSRoles: A role-driven authority for ds-auth which facilitates access to lists of user roles and capabilities. Works as a set of lookup tables for the canCall function to provide boolean answers as to whether a user is authorized to call a given function at given address. DSRoles provides 3 different ways of permitting/forbidding function call access to users:1) Root Users: any users added to the _root_users whitelist will be authorized to call any function regardless of what roles or capabilities might be defined. 2) Public Capabilities: public capabilities are global capabilities which apply to all users and take precedence over any user specific role-capabilities which might be defined. 3) Role Capabilities: capabilities which are associated with a particular role. Role capabilities are only checked if the user does not have root access and the capability is not public.

WETH: WETH stands for Wrapped Ether. For protocols that work with ERC-20 tokens but also need to handle Ether, WETH contracts allow converting Ether to its ERC-20 equivalent WETH (called wrapping) and vice-versa (called unwrapping). WETH can be created by sending ether to a WETH smart contract where the Ether is stored and in turn receiving the WETH ERC-20 token at a 1:1 ratio. This WETH can be sent back to the same smart contract to be "unwrapped" i.e. redeemed back for the original Ether at a 1:1 ratio. The most widely used WETH contract is WETH9 which holds more than 7 million Ether for now.

Uniswap V2: Uniswap is an automated liquidity protocol powered by a constant product formula and implemented in a system of non-upgradeable smart contracts on the Ethereum blockchain. The automated market making algorithm used by Uniswap is $x*y=k$, where x and y represent a token pair that allow one token to be exchanged for the other as long as the "constant product" formula is preserved i.e. trades must not change the product (k) of a pair's reserve balances (x and y). Core concepts:

a. Pools: Each Uniswap liquidity pool is a trading venue for a pair of ERC20 tokens. When a pool contract is created, its balances of each token are 0; in order for the pool to begin facilitating trades, someone must seed it with an initial deposit of each token. This first liquidity provider is the one who sets the initial price of the pool. They are incentivized to deposit an equal value of both tokens into the pool. Whenever liquidity is deposited into a pool, unique tokens known as liquidity tokens are minted and sent to the provider's address. These tokens represent a given liquidity provider's contribution to a pool.

b. Swaps: allows one to trade one ERC-20 token for another, where one token is withdrawn (purchased) and a proportional amount of the other deposited (sold), in order to maintain the constant $x*y=k$

c. Flash Swaps: allows one to withdraw up to the full reserves of any ERC20 token on Uniswap and execute arbitrary logic at no upfront cost, provided that by the end of the transaction they either: 1) pay for the withdrawn ERC20 tokens with the corresponding pair tokens 2) return the withdrawn ERC20 tokens along with a small fee

d. Oracles: enables developers to build highly decentralized and manipulation-resistant on-chain price oracles. A price oracle is any tool used to view price information about a given asset. Every pair measures (but does not store) the market price at the beginning of each block, before any trades take place i.e. price at the end of the previous block which is added to a single cumulative-price variable weighted by the amount of time this price existed. This variable can be used by external contracts to track accurate time-weighted average prices (TWAPs) across any time interval.

Uniswap V3: Introduces

a. Concentrated liquidity: giving individual LPs granular control over what price ranges their capital is allocated to. Individual positions are aggregated together into a single pool, forming one combined curve for users to trade against

b. Multiple fee tiers: allowing LPs to be appropriately compensated for taking on varying degrees of risk

c. V3 oracles are capable of providing time-weighted average prices (TWAPs) on demand for any period within the last ~9 days. This removes the need for integrators to checkpoint historical values.

Chainlink Oracles & Price Feeds: Chainlink Price Feeds provide aggregated data (via its *AggregatorV3Interface* contract interface) from various high quality data providers, fed on-chain by decentralized oracles on the Chainlink Network. To get price data into smart contracts for an asset that isn't covered by an existing price feed, such as the price of a particular stock, one can customize Chainlink oracles to call any external API.