// SPDX-License-Identifier: MIT

// File: @openzeppelin/contracts/token/ERC20/IERC20.sol

// OpenZeppelin Contracts (last updated v5.0.0) (token/ERC20/IERC20.sol)

pragma solidity ^0.8.20;

/\*\*

\* @dev Interface of the ERC20 standard as defined in the EIP.

\*/

interface IERC20 {

/\*\*

\* @dev Emitted when `value` tokens are moved from one account (`from`) to

\* another (`to`).

\*

\* Note that `value` may be zero.

\*/

event Transfer(address indexed from, address indexed to, uint256 value);

/\*\*

\* @dev Emitted when the allowance of a `spender` for an `owner` is set by

\* a call to {approve}. `value` is the new allowance.

\*/

event Approval(address indexed owner, address indexed spender, uint256 value);

/\*\*

\* @dev Returns the value of tokens in existence.

\*/

function totalSupply() external view returns (uint256);

/\*\*

\* @dev Returns the value of tokens owned by `account`.

\*/

function balanceOf(address account) external view returns (uint256);

/\*\*

\* @dev Moves a `value` amount of tokens from the caller's account to `to`.

\*

\* Returns a boolean value indicating whether the operation succeeded.

\*

\* Emits a {Transfer} event.

\*/

function transfer(address to, uint256 value) external returns (bool);

/\*\*

\* @dev 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.

\*/

function allowance(address owner, address spender) external view returns (uint256);

/\*\*

\* @dev Sets a `value` amount of tokens as the allowance of `spender` over the

\* caller's tokens.

\*

\* Returns a boolean value indicating whether the operation succeeded.

\*

\* IMPORTANT: Beware that 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

\*

\* Emits an {Approval} event.

\*/

function approve(address spender, uint256 value) external returns (bool);

/\*\*

\* @dev Moves a `value` amount of tokens from `from` to `to` using the

\* allowance mechanism. `value` is then deducted from the caller's

\* allowance.

\*

\* Returns a boolean value indicating whether the operation succeeded.

\*

\* Emits a {Transfer} event.

\*/

function transferFrom(address from, address to, uint256 value) external returns (bool);

}

// File: @openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol

// OpenZeppelin Contracts (last updated v5.0.0) (token/ERC20/extensions/IERC20Metadata.sol)

pragma solidity ^0.8.20;

/\*\*

\* @dev Interface for the optional metadata functions from the ERC20 standard.

\*/

interface IERC20Metadata is IERC20 {

/\*\*

\* @dev Returns the name of the token.

\*/

function name() external view returns (string memory);

/\*\*

\* @dev Returns the symbol of the token.

\*/

function symbol() external view returns (string memory);

/\*\*

\* @dev Returns the decimals places of the token.

\*/

function decimals() external view returns (uint8);

}

// File: @openzeppelin/contracts/utils/Context.sol

// OpenZeppelin Contracts (last updated v5.0.1) (utils/Context.sol)

pragma solidity ^0.8.20;

/\*\*

\* @dev 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.

\*/

abstract contract Context {

function \_msgSender() internal view virtual returns (address) {

return msg.sender;

}

function \_msgData() internal view virtual returns (bytes calldata) {

return msg.data;

}

function \_contextSuffixLength() internal view virtual returns (uint256) {

return 0;

}

}

// File: @openzeppelin/contracts/interfaces/draft-IERC6093.sol

// OpenZeppelin Contracts (last updated v5.0.0) (interfaces/draft-IERC6093.sol)

pragma solidity ^0.8.20;

/\*\*

\* @dev Standard ERC20 Errors

\* Interface of the https://eips.ethereum.org/EIPS/eip-6093[ERC-6093] custom errors for ERC20 tokens.

\*/

interface IERC20Errors {

/\*\*

\* @dev Indicates an error related to the current `balance` of a `sender`. Used in transfers.

\* @param sender Address whose tokens are being transferred.

\* @param balance Current balance for the interacting account.

\* @param needed Minimum amount required to perform a transfer.

\*/

error ERC20InsufficientBalance(address sender, uint256 balance, uint256 needed);

/\*\*

\* @dev Indicates a failure with the token `sender`. Used in transfers.

\* @param sender Address whose tokens are being transferred.

\*/

error ERC20InvalidSender(address sender);

/\*\*

\* @dev Indicates a failure with the token `receiver`. Used in transfers.

\* @param receiver Address to which tokens are being transferred.

\*/

error ERC20InvalidReceiver(address receiver);

/\*\*

\* @dev Indicates a failure with the `spender`’s `allowance`. Used in transfers.

\* @param spender Address that may be allowed to operate on tokens without being their owner.

\* @param allowance Amount of tokens a `spender` is allowed to operate with.

\* @param needed Minimum amount required to perform a transfer.

\*/

error ERC20InsufficientAllowance(address spender, uint256 allowance, uint256 needed);

/\*\*

\* @dev Indicates a failure with the `approver` of a token to be approved. Used in approvals.

\* @param approver Address initiating an approval operation.

\*/

error ERC20InvalidApprover(address approver);

/\*\*

\* @dev Indicates a failure with the `spender` to be approved. Used in approvals.

\* @param spender Address that may be allowed to operate on tokens without being their owner.

\*/

error ERC20InvalidSpender(address spender);

}

/\*\*

\* @dev Standard ERC721 Errors

\* Interface of the https://eips.ethereum.org/EIPS/eip-6093[ERC-6093] custom errors for ERC721 tokens.

\*/

interface IERC721Errors {

/\*\*

\* @dev Indicates that an address can't be an owner. For example, `address(0)` is a forbidden owner in EIP-20.

\* Used in balance queries.

\* @param owner Address of the current owner of a token.

\*/

error ERC721InvalidOwner(address owner);

/\*\*

\* @dev Indicates a `tokenId` whose `owner` is the zero address.

\* @param tokenId Identifier number of a token.

\*/

error ERC721NonexistentToken(uint256 tokenId);

/\*\*

\* @dev Indicates an error related to the ownership over a particular token. Used in transfers.

\* @param sender Address whose tokens are being transferred.

\* @param tokenId Identifier number of a token.

\* @param owner Address of the current owner of a token.

\*/

error ERC721IncorrectOwner(address sender, uint256 tokenId, address owner);

/\*\*

\* @dev Indicates a failure with the token `sender`. Used in transfers.

\* @param sender Address whose tokens are being transferred.

\*/

error ERC721InvalidSender(address sender);

/\*\*

\* @dev Indicates a failure with the token `receiver`. Used in transfers.

\* @param receiver Address to which tokens are being transferred.

\*/

error ERC721InvalidReceiver(address receiver);

/\*\*

\* @dev Indicates a failure with the `operator`’s approval. Used in transfers.

\* @param operator Address that may be allowed to operate on tokens without being their owner.

\* @param tokenId Identifier number of a token.

\*/

error ERC721InsufficientApproval(address operator, uint256 tokenId);

/\*\*

\* @dev Indicates a failure with the `approver` of a token to be approved. Used in approvals.

\* @param approver Address initiating an approval operation.

\*/

error ERC721InvalidApprover(address approver);

/\*\*

\* @dev Indicates a failure with the `operator` to be approved. Used in approvals.

\* @param operator Address that may be allowed to operate on tokens without being their owner.

\*/

error ERC721InvalidOperator(address operator);

}

/\*\*

\* @dev Standard ERC1155 Errors

\* Interface of the https://eips.ethereum.org/EIPS/eip-6093[ERC-6093] custom errors for ERC1155 tokens.

\*/

interface IERC1155Errors {

/\*\*

\* @dev Indicates an error related to the current `balance` of a `sender`. Used in transfers.

\* @param sender Address whose tokens are being transferred.

\* @param balance Current balance for the interacting account.

\* @param needed Minimum amount required to perform a transfer.

\* @param tokenId Identifier number of a token.

\*/

error ERC1155InsufficientBalance(address sender, uint256 balance, uint256 needed, uint256 tokenId);

/\*\*

\* @dev Indicates a failure with the token `sender`. Used in transfers.

\* @param sender Address whose tokens are being transferred.

\*/

error ERC1155InvalidSender(address sender);

/\*\*

\* @dev Indicates a failure with the token `receiver`. Used in transfers.

\* @param receiver Address to which tokens are being transferred.

\*/

error ERC1155InvalidReceiver(address receiver);

/\*\*

\* @dev Indicates a failure with the `operator`’s approval. Used in transfers.

\* @param operator Address that may be allowed to operate on tokens without being their owner.

\* @param owner Address of the current owner of a token.

\*/

error ERC1155MissingApprovalForAll(address operator, address owner);

/\*\*

\* @dev Indicates a failure with the `approver` of a token to be approved. Used in approvals.

\* @param approver Address initiating an approval operation.

\*/

error ERC1155InvalidApprover(address approver);

/\*\*

\* @dev Indicates a failure with the `operator` to be approved. Used in approvals.

\* @param operator Address that may be allowed to operate on tokens without being their owner.

\*/

error ERC1155InvalidOperator(address operator);

/\*\*

\* @dev Indicates an array length mismatch between ids and values in a safeBatchTransferFrom operation.

\* Used in batch transfers.

\* @param idsLength Length of the array of token identifiers

\* @param valuesLength Length of the array of token amounts

\*/

error ERC1155InvalidArrayLength(uint256 idsLength, uint256 valuesLength);

}

// File: @openzeppelin/contracts/token/ERC20/ERC20.sol

// OpenZeppelin Contracts (last updated v5.0.0) (token/ERC20/ERC20.sol)

pragma solidity ^0.8.20;

/\*\*

\* @dev Implementation of the {IERC20} interface.

\*

\* This implementation is agnostic to the way tokens are created. This means

\* that a supply mechanism has to be added in a derived contract using {\_mint}.

\*

\* TIP: For a detailed writeup see our guide

\* https://forum.openzeppelin.com/t/how-to-implement-erc20-supply-mechanisms/226[How

\* to implement supply mechanisms].

\*

\* The default value of {decimals} is 18. To change this, you should override

\* this function so it returns a different value.

\*

\* We have followed general OpenZeppelin Contracts guidelines: functions revert

\* instead returning `false` on failure. This behavior is nonetheless

\* conventional and does not conflict with the expectations of ERC20

\* applications.

\*

\* Additionally, an {Approval} event is emitted on calls to {transferFrom}.

\* This allows applications to reconstruct the allowance for all accounts just

\* by listening to said events. Other implementations of the EIP may not emit

\* these events, as it isn't required by the specification.

\*/

abstract contract ERC20 is Context, IERC20, IERC20Metadata, IERC20Errors {

mapping(address account => uint256) private \_balances;

mapping(address account => mapping(address spender => uint256)) private \_allowances;

uint256 private \_totalSupply;

string private \_name;

string private \_symbol;

/\*\*

\* @dev Sets the values for {name} and {symbol}.

\*

\* All two of these values are immutable: they can only be set once during

\* construction.

\*/

constructor(string memory name\_, string memory symbol\_) {

\_name = name\_;

\_symbol = symbol\_;

}

/\*\*

\* @dev Returns the name of the token.

\*/

function name() public view virtual returns (string memory) {

return \_name;

}

/\*\*

\* @dev Returns the symbol of the token, usually a shorter version of the

\* name.

\*/

function symbol() public view virtual returns (string memory) {

return \_symbol;

}

/\*\*

\* @dev 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 default value returned by this function, unless

\* it's overridden.

\*

\* NOTE: This information is only used for \_display\_ purposes: it in

\* no way affects any of the arithmetic of the contract, including

\* {IERC20-balanceOf} and {IERC20-transfer}.

\*/

function decimals() public view virtual returns (uint8) {

return 18;

}

/\*\*

\* @dev See {IERC20-totalSupply}.

\*/

function totalSupply() public view virtual returns (uint256) {

return \_totalSupply;

}

/\*\*

\* @dev See {IERC20-balanceOf}.

\*/

function balanceOf(address account) public view virtual returns (uint256) {

return \_balances[account];

}

/\*\*

\* @dev See {IERC20-transfer}.

\*

\* Requirements:

\*

\* - `to` cannot be the zero address.

\* - the caller must have a balance of at least `value`.

\*/

function transfer(address to, uint256 value) public virtual returns (bool) {

address owner = \_msgSender();

\_transfer(owner, to, value);

return true;

}

/\*\*

\* @dev See {IERC20-allowance}.

\*/

function allowance(address owner, address spender) public view virtual returns (uint256) {

return \_allowances[owner][spender];

}

/\*\*

\* @dev See {IERC20-approve}.

\*

\* NOTE: If `value` is the maximum `uint256`, the allowance is not updated on

\* `transferFrom`. This is semantically equivalent to an infinite approval.

\*

\* Requirements:

\*

\* - `spender` cannot be the zero address.

\*/

function approve(address spender, uint256 value) public virtual returns (bool) {

address owner = \_msgSender();

\_approve(owner, spender, value);

return true;

}

/\*\*

\* @dev See {IERC20-transferFrom}.

\*

\* Emits an {Approval} event indicating the updated allowance. This is not

\* required by the EIP. See the note at the beginning of {ERC20}.

\*

\* NOTE: Does not update the allowance if the current allowance

\* is the maximum `uint256`.

\*

\* Requirements:

\*

\* - `from` and `to` cannot be the zero address.

\* - `from` must have a balance of at least `value`.

\* - the caller must have allowance for ``from``'s tokens of at least

\* `value`.

\*/

function transferFrom(address from, address to, uint256 value) public virtual returns (bool) {

address spender = \_msgSender();

\_spendAllowance(from, spender, value);

\_transfer(from, to, value);

return true;

}

/\*\*

\* @dev Moves a `value` amount of tokens from `from` to `to`.

\*

\* This internal function is equivalent to {transfer}, and can be used to

\* e.g. implement automatic token fees, slashing mechanisms, etc.

\*

\* Emits a {Transfer} event.

\*

\* NOTE: This function is not virtual, {\_update} should be overridden instead.

\*/

function \_transfer(address from, address to, uint256 value) internal {

if (from == address(0)) {

revert ERC20InvalidSender(address(0));

}

if (to == address(0)) {

revert ERC20InvalidReceiver(address(0));

}

\_update(from, to, value);

}

/\*\*

\* @dev Transfers a `value` amount of tokens from `from` to `to`, or alternatively mints (or burns) if `from`

\* (or `to`) is the zero address. All customizations to transfers, mints, and burns should be done by overriding

\* this function.

\*

\* Emits a {Transfer} event.

\*/

function \_update(address from, address to, uint256 value) internal virtual {

if (from == address(0)) {

// Overflow check required: The rest of the code assumes that totalSupply never overflows

\_totalSupply += value;

} else {

uint256 fromBalance = \_balances[from];

if (fromBalance < value) {

revert ERC20InsufficientBalance(from, fromBalance, value);

}

unchecked {

// Overflow not possible: value <= fromBalance <= totalSupply.

\_balances[from] = fromBalance - value;

}

}

if (to == address(0)) {

unchecked {

// Overflow not possible: value <= totalSupply or value <= fromBalance <= totalSupply.

\_totalSupply -= value;

}

} else {

unchecked {

// Overflow not possible: balance + value is at most totalSupply, which we know fits into a uint256.

\_balances[to] += value;

}

}

emit Transfer(from, to, value);

}

/\*\*

\* @dev Creates a `value` amount of tokens and assigns them to `account`, by transferring it from address(0).

\* Relies on the `\_update` mechanism

\*

\* Emits a {Transfer} event with `from` set to the zero address.

\*

\* NOTE: This function is not virtual, {\_update} should be overridden instead.

\*/

function \_mint(address account, uint256 value) internal {

if (account == address(0)) {

revert ERC20InvalidReceiver(address(0));

}

\_update(address(0), account, value);

}

/\*\*

\* @dev Destroys a `value` amount of tokens from `account`, lowering the total supply.

\* Relies on the `\_update` mechanism.

\*

\* Emits a {Transfer} event with `to` set to the zero address.

\*

\* NOTE: This function is not virtual, {\_update} should be overridden instead

\*/

function \_burn(address account, uint256 value) internal {

if (account == address(0)) {

revert ERC20InvalidSender(address(0));

}

\_update(account, address(0), value);

}

/\*\*

\* @dev Sets `value` as the allowance of `spender` over the `owner` s tokens.

\*

\* This internal function is equivalent to `approve`, and can be used to

\* e.g. set automatic allowances for certain subsystems, etc.

\*

\* Emits an {Approval} event.

\*

\* Requirements:

\*

\* - `owner` cannot be the zero address.

\* - `spender` cannot be the zero address.

\*

\* Overrides to this logic should be done to the variant with an additional `bool emitEvent` argument.

\*/

function \_approve(address owner, address spender, uint256 value) internal {

\_approve(owner, spender, value, true);

}

/\*\*

\* @dev Variant of {\_approve} with an optional flag to enable or disable the {Approval} event.

\*

\* By default (when calling {\_approve}) the flag is set to true. On the other hand, approval changes made by

\* `\_spendAllowance` during the `transferFrom` operation set the flag to false. This saves gas by not emitting any

\* `Approval` event during `transferFrom` operations.

\*

\* Anyone who wishes to continue emitting `Approval` events on the`transferFrom` operation can force the flag to

\* true using the following override:

\* ```

\* function \_approve(address owner, address spender, uint256 value, bool) internal virtual override {

\* super.\_approve(owner, spender, value, true);

\* }

\* ```

\*

\* Requirements are the same as {\_approve}.

\*/

function \_approve(address owner, address spender, uint256 value, bool emitEvent) internal virtual {

if (owner == address(0)) {

revert ERC20InvalidApprover(address(0));

}

if (spender == address(0)) {

revert ERC20InvalidSpender(address(0));

}

\_allowances[owner][spender] = value;

if (emitEvent) {

emit Approval(owner, spender, value);

}

}

/\*\*

\* @dev Updates `owner` s allowance for `spender` based on spent `value`.

\*

\* Does not update the allowance value in case of infinite allowance.

\* Revert if not enough allowance is available.

\*

\* Does not emit an {Approval} event.

\*/

function \_spendAllowance(address owner, address spender, uint256 value) internal virtual {

uint256 currentAllowance = allowance(owner, spender);

if (currentAllowance != type(uint256).max) {

if (currentAllowance < value) {

revert ERC20InsufficientAllowance(spender, currentAllowance, value);

}

unchecked {

\_approve(owner, spender, currentAllowance - value, false);

}

}

}

}

// File: ERC201.sol

pragma solidity ^0.8.24;

// TONCONTRAT ERC20 Token Contract

contract TONCONTRAT is ERC20 {

// Constructor that mints the initial supply to the deployer's address

constructor(uint256 initialSupply) ERC20("jlerler", "JLER") {

\_mint(msg.sender, initialSupply); // Mint the initial supply of tokens to the deployer's address

}

// Function to mint new tokens to a specified address

function mint(address to, uint256 amount) public {

\_mint(to, amount); // Call internal \_mint function to create new tokens

}

// Function to burn tokens from a specified address

function burn(address from, uint256 amount) public {

\_burn(from, amount); // Call internal \_burn function to destroy tokens

}

// Override the transfer function to allow sending tokens

function transfer(address to, uint256 amount) public override returns (bool) {

\_transfer(\_msgSender(), to, amount); // Call internal \_transfer function

return true; // Return success

}

// Override the approve function to allow an address to spend on behalf of the caller

function approve(address spender, uint256 amount) public override returns (bool) {

\_approve(\_msgSender(), spender, amount); // Call internal \_approve function

return true; // Return success

}

// Override the transferFrom function to allow spending tokens on behalf of an address

function transferFrom(address from, address to, uint256 amount) public override returns (bool) {

\_transfer(from, to, amount); // Transfer tokens from `from` to `to`

uint256 currentAllowance = allowance(from, \_msgSender());

require(currentAllowance >= amount, "ERC20: transfer amount exceeds allowance");

\_approve(from, \_msgSender(), currentAllowance - amount); // Update allowance

return true; // Return success

}

// Function to retrieve the balance of a specific address

function getBalanceOf(address account) public view returns (uint256) {

return balanceOf(account); // Call balanceOf to get the balance

}

}