**Description**

**Story of the Arbiter**

Villains band together and cause enough destruction in the Multiverse that it forces the last cosmic deity to take notice and intervene. With all heroes either badly defeated or wiped completely out, darkness is quickly on the rise. A cosmic deity descends to the creation planet known as **SOURCE WORLD** in a great beam of light to confront his enemies. He is...The Arbiter, a massive being of solar light and immense power. His giant body postures bravely in the midst of this dying world. Their to make his last stand for a system of universes that cannot stand for themselves.

His enemies converge on him. Their sinister leader quickly dispatches his treacherous minions to battle the cosmic deity. The battle proves too much for the deity and he is ended. As he dies his body hardens into a crystalline form, suddenly explodes and into billions of pieces.

The Multiverse is forever altered and now depends on you to continue its story!

**Description of contract code**

The Arbiter Token is an ERC20 standard token that is capped to 4,444,444,444. The ARBTERCToken contract specifies the total supply, the market cap and further token economics. The TokenSale contract inherits from the ARBTERC20Token contract and implements the functionality for initial token offerings.

The setPrice() function allows the owner of the contract to manipulate the price of the Arbiter Token. This function cannot be called by anyone else other than the owner.

The bookTokens() function allows any investor to book tokens under their address by depositing the appropriate amount.

The addTokens() function allows the owner to mint more of the Arbiter tokens.

The withdrawBooking() function allows the investors to withdraw their booking of some or all tokens before the token allocation has been done.

The allocateBookedTokens() function allows owner to allocate all the tokens booked by the investors respectively.

**CONTRACT CODE**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

/\*

 \* @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 GSN 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 payable) {

        return payable(msg.sender);

    }

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

        this; // silence state mutability warning without generating bytecode - see https://github.com/ethereum/solidity/issues/2691

        return msg.data;

    }

}

// File: @openzeppelin/contracts/access/Ownable.sol

/\*\*

 \* @dev Contract module which 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.

 \*/

abstract contract Ownable is Context {

    address private \_owner;

    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

    /\*\*

     \* @dev Initializes the contract setting the deployer as the initial owner.

     \*/

    constructor () {

        address msgSender = \_msgSender();

        \_owner = msgSender;

        emit OwnershipTransferred(address(0), msgSender);

    }

    /\*\*

     \* @dev Returns the address of the current owner.

     \*/

    function owner() public view returns (address) {

        return \_owner;

    }

    /\*\*

     \* @dev Throws if called by any account other than the owner.

     \*/

    modifier onlyOwner() {

        require(\_owner == \_msgSender(), "Ownable: caller is not the owner");

        \_;

    }

    /\*\*

     \* @dev Leaves the contract without owner. It will not be possible to call

     \* `onlyOwner` functions anymore. Can only be called by the current owner.

     \*

     \* NOTE: Renouncing ownership will leave the contract without an owner,

     \* thereby removing any functionality that is only available to the owner.

     \*/

    function renounceOwnership() public virtual onlyOwner {

        emit OwnershipTransferred(\_owner, address(0));

        \_owner = address(0);

    }

    /\*\*

     \* @dev Transfers ownership of the contract to a new account (`newOwner`).

     \* Can only be called by the current owner.

     \*/

    function transferOwnership(address newOwner) public virtual onlyOwner {

        require(newOwner != address(0), "Ownable: new owner is the zero address");

        emit OwnershipTransferred(\_owner, newOwner);

        \_owner = newOwner;

    }

}

/\*\*

 \* @title Pausable

 \* @dev Base contract which allows children to implement an emergency stop mechanism.

 \*/

contract Pausable is Ownable {

  event Pause();

  event Unpause();

  event NotPausable();

  bool public paused = false;

  bool public canPause = true;

  /\*\*

   \* @dev Modifier to make a function callable only when the contract is not paused.

   \*/

  modifier whenNotPaused() {

    require(!paused || msg.sender == owner());

    \_;

  }

  /\*\*

   \* @dev Modifier to make a function callable only when the contract is paused.

   \*/

  modifier whenPaused() {

    require(paused);

    \_;

  }

  /\*\*

     \* @dev called by the owner to pause, triggers stopped state

     \*\*/

function pause() onlyOwner whenNotPaused public {

        require(canPause == true);

        paused = true;

        emit Pause();

    }

  /\*\*

   \* @dev called by the owner to unpause, returns to normal state

   \*/

  function unpause() onlyOwner whenPaused public {

    require(paused == true);

    paused = false;

    emit Unpause();

  }

  /\*\*

     \* @dev Prevent the token from ever being paused again

     \*\*/

    function notPausable() onlyOwner public{

        paused = false;

        canPause = false;

        emit NotPausable();

    }

}

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

/\*\*

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

 \*/

interface IERC20 {

    /\*\*

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

     \*/

    function totalSupply() external view returns (uint256);

    /\*\*

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

     \*/

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

    /\*\*

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

     \*

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

     \*

     \* Emits a {Transfer} event.

     \*/

    function transfer(address recipient, uint256 amount) 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 `amount` 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 amount) external returns (bool);

    /\*\*

     \* @dev 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.

     \*/

    function transferFrom(address sender, address recipient, uint256 amount) external returns (bool);

    /\*\*

     \* @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);

}

// File: @openzeppelin/contracts/math/SafeMath.sol

/\*\*

 \* @dev Wrappers over Solidity's arithmetic operations with added overflow

 \* checks.

 \*

 \* Arithmetic operations in Solidity wrap on overflow. This can easily result

 \* in bugs, because programmers usually assume that an overflow raises an

 \* error, which is the standard behavior in high level programming languages.

 \* `SafeMath` restores this intuition by reverting the transaction when an

 \* operation overflows.

 \*

 \* Using this library instead of the unchecked operations eliminates an entire

 \* class of bugs, so it's recommended to use it always.

 \*/

library SafeMath {

    /\*\*

     \* @dev Returns the addition of two unsigned integers, reverting on

     \* overflow.

     \*

     \* Counterpart to Solidity's `+` operator.

     \*

     \* Requirements:

     \*

     \* - Addition cannot overflow.

     \*/

    function add(uint256 a, uint256 b) internal pure returns (uint256) {

        uint256 c = a + b;

        require(c >= a, "SafeMath: addition overflow");

        return c;

    }

    /\*\*

     \* @dev Returns the subtraction of two unsigned integers, reverting on

     \* overflow (when the result is negative).

     \*

     \* Counterpart to Solidity's `-` operator.

     \*

     \* Requirements:

     \*

     \* - Subtraction cannot overflow.

     \*/

    function sub(uint256 a, uint256 b) internal pure returns (uint256) {

        return sub(a, b, "SafeMath: subtraction overflow");

    }

    /\*\*

     \* @dev Returns the subtraction of two unsigned integers, reverting with custom message on

     \* overflow (when the result is negative).

     \*

     \* Counterpart to Solidity's `-` operator.

     \*

     \* Requirements:

     \*

     \* - Subtraction cannot overflow.

     \*/

    function sub(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {

        require(b <= a, errorMessage);

        uint256 c = a - b;

        return c;

    }

    /\*\*

     \* @dev Returns the multiplication of two unsigned integers, reverting on

     \* overflow.

     \*

     \* Counterpart to Solidity's `\*` operator.

     \*

     \* Requirements:

     \*

     \* - Multiplication cannot overflow.

     \*/

    function mul(uint256 a, uint256 b) internal pure returns (uint256) {

        // Gas optimization: this is cheaper than requiring 'a' not being zero, but the

        // benefit is lost if 'b' is also tested.

        // See: https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522

        if (a == 0) {

            return 0;

        }

        uint256 c = a \* b;

        require(c / a == b, "SafeMath: multiplication overflow");

        return c;

    }

    /\*\*

     \* @dev Returns the integer division of two unsigned integers. Reverts on

     \* division by zero. The result is rounded towards zero.

     \*

     \* Counterpart to Solidity's `/` operator. Note: this function uses a

     \* `revert` opcode (which leaves remaining gas untouched) while Solidity

     \* uses an invalid opcode to revert (consuming all remaining gas).

     \*

     \* Requirements:

     \*

     \* - The divisor cannot be zero.

     \*/

    function div(uint256 a, uint256 b) internal pure returns (uint256) {

        return div(a, b, "SafeMath: division by zero");

    }

    /\*\*

     \* @dev Returns the integer division of two unsigned integers. Reverts with custom message on

     \* division by zero. The result is rounded towards zero.

     \*

     \* Counterpart to Solidity's `/` operator. Note: this function uses a

     \* `revert` opcode (which leaves remaining gas untouched) while Solidity

     \* uses an invalid opcode to revert (consuming all remaining gas).

     \*

     \* Requirements:

     \*

     \* - The divisor cannot be zero.

     \*/

    function div(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {

        require(b > 0, errorMessage);

        uint256 c = a / b;

        // assert(a == b \* c + a % b); // There is no case in which this doesn't hold

        return c;

    }

    /\*\*

     \* @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),

     \* Reverts when dividing by zero.

     \*

     \* Counterpart to Solidity's `%` operator. This function uses a `revert`

     \* opcode (which leaves remaining gas untouched) while Solidity uses an

     \* invalid opcode to revert (consuming all remaining gas).

     \*

     \* Requirements:

     \*

     \* - The divisor cannot be zero.

     \*/

    function mod(uint256 a, uint256 b) internal pure returns (uint256) {

        return mod(a, b, "SafeMath: modulo by zero");

    }

    /\*\*

     \* @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),

     \* Reverts with custom message when dividing by zero.

     \*

     \* Counterpart to Solidity's `%` operator. This function uses a `revert`

     \* opcode (which leaves remaining gas untouched) while Solidity uses an

     \* invalid opcode to revert (consuming all remaining gas).

     \*

     \* Requirements:

     \*

     \* - The divisor cannot be zero.

     \*/

    function mod(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {

        require(b != 0, errorMessage);

        return a % b;

    }

}

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

/\*\*

 \* @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}.

 \* For a generic mechanism see {ERC20PresetMinterPauser}.

 \*

 \* TIP: For a detailed writeup see our guide

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

 \* to implement supply mechanisms].

 \*

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

 \* of 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.

 \*

 \* Finally, the non-standard {decreaseAllowance} and {increaseAllowance}

 \* functions have been added to mitigate the well-known issues around setting

 \* allowances. See {IERC20-approve}.

 \*/

contract ERC20 is Context, IERC20,Pausable {

    using SafeMath for uint256;

    mapping (address => uint256) private \_balances;

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

    uint256 private \_totalSupply;

    string private \_name;

    string private \_symbol;

    uint8 private \_decimals;

    /\*\*

     \* @dev Sets the values for {name} and {symbol}, initializes {decimals} with

     \* a default value of 18.

     \*

     \* To select a different value for {decimals}, use {\_setupDecimals}.

     \*

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

     \* construction.

     \*/

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

        \_name = name\_;

        \_symbol = symbol\_;

        \_decimals = 18;

    }

    /\*\*

     \* @dev Returns the name of the token.

     \*/

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

        return \_name;

    }

    /\*\*

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

     \* name.

     \*/

    function symbol() public view 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 value {ERC20} uses, unless {\_setupDecimals} is

     \* called.

     \*

     \* 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 returns (uint8) {

        return \_decimals;

    }

    /\*\*

     \* @dev See {IERC20-totalSupply}.

     \*/

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

        return \_totalSupply;

    }

    /\*\*

     \* @dev See {IERC20-balanceOf}.

     \*/

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

        return \_balances[account];

    }

    /\*\*

     \* @dev See {IERC20-transfer}.

     \*

     \* Requirements:

     \*

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

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

     \*/

    function transfer(address recipient, uint256 amount) public whenNotPaused virtual override returns (bool) {

        \_transfer(\_msgSender(), recipient, amount);

        return true;

    }

    /\*\*

     \* @dev See {IERC20-allowance}.

     \*/

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

        return \_allowances[owner][spender];

    }

    /\*\*

     \* @dev See {IERC20-approve}.

     \*

     \* Requirements:

     \*

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

     \*/

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

        \_approve(\_msgSender(), spender, amount);

        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}.

     \*

     \* Requirements:

     \*

     \* - `sender` and `recipient` cannot be the zero address.

     \* - `sender` must have a balance of at least `amount`.

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

     \* `amount`.

     \*/

    function transferFrom(address sender, address recipient, uint256 amount) public whenNotPaused virtual override returns (bool) {

        \_transfer(sender, recipient, amount);

        \_approve(sender, \_msgSender(), \_allowances[sender][\_msgSender()].sub(amount, "ERC20: transfer amount exceeds allowance"));

        return true;

    }

    /\*\*

     \* @dev Atomically increases the allowance granted to `spender` by the caller.

     \*

     \* This is an alternative to {approve} that can be used as a mitigation for

     \* problems described in {IERC20-approve}.

     \*

     \* Emits an {Approval} event indicating the updated allowance.

     \*

     \* Requirements:

     \*

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

     \*/

    function increaseAllowance(address spender, uint256 addedValue) public whenNotPaused virtual returns (bool) {

        \_approve(\_msgSender(), spender, \_allowances[\_msgSender()][spender].add(addedValue));

        return true;

    }

    /\*\*

     \* @dev Atomically decreases the allowance granted to `spender` by the caller.

     \*

     \* This is an alternative to {approve} that can be used as a mitigation for

     \* problems described in {IERC20-approve}.

     \*

     \* Emits an {Approval} event indicating the updated allowance.

     \*

     \* Requirements:

     \*

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

     \* - `spender` must have allowance for the caller of at least

     \* `subtractedValue`.

     \*/

    function decreaseAllowance(address spender, uint256 subtractedValue) public whenNotPaused virtual returns (bool) {

        \_approve(\_msgSender(), spender, \_allowances[\_msgSender()][spender].sub(subtractedValue, "ERC20: decreased allowance below zero"));

        return true;

    }

    /\*\*

     \* @dev Moves tokens `amount` from `sender` to `recipient`.

     \*

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

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

     \*

     \* Emits a {Transfer} event.

     \*

     \* Requirements:

     \*

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

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

     \* - `sender` must have a balance of at least `amount`.

     \*/

    function \_transfer(address sender, address recipient, uint256 amount) internal virtual {

        require(sender != address(0), "ERC20: transfer from the zero address");

        require(recipient != address(0), "ERC20: transfer to the zero address");

        \_beforeTokenTransfer(sender, recipient, amount);

        \_balances[sender] = \_balances[sender].sub(amount, "ERC20: transfer amount exceeds balance");

        \_balances[recipient] = \_balances[recipient].add(amount);

        emit Transfer(sender, recipient, amount);

    }

    /\*\* @dev Creates `amount` tokens and assigns them to `account`, increasing

     \* the total supply.

     \*

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

     \*

     \* Requirements:

     \*

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

     \*/

    function \_mint(address account, uint256 amount) internal virtual {

        require(account != address(0), "ERC20: mint to the zero address");

        \_beforeTokenTransfer(address(0), account, amount);

        \_totalSupply = \_totalSupply.add(amount);

        \_balances[account] = \_balances[account].add(amount);

        emit Transfer(address(0), account, amount);

    }

    /\*\*

     \* @dev Destroys `amount` tokens from `account`, reducing the

     \* total supply.

     \*

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

     \*

     \* Requirements:

     \*

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

     \* - `account` must have at least `amount` tokens.

     \*/

    function \_burn(address account, uint256 amount) internal whenNotPaused virtual {

        require(account != address(0), "ERC20: burn from the zero address");

        \_beforeTokenTransfer(account, address(0), amount);

        \_balances[account] = \_balances[account].sub(amount, "ERC20: burn amount exceeds balance");

        \_totalSupply = \_totalSupply.sub(amount);

        emit Transfer(account, address(0), amount);

    }

    /\*\*

     \* @dev Sets `amount` 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.

     \*/

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

        require(owner != address(0), "ERC20: approve from the zero address");

        require(spender != address(0), "ERC20: approve to the zero address");

        \_allowances[owner][spender] = amount;

        emit Approval(owner, spender, amount);

    }

    /\*\*

     \* @dev Sets {decimals} to a value other than the default one of 18.

     \*

     \* WARNING: This function should only be called from the constructor. Most

     \* applications that interact with token contracts will not expect

     \* {decimals} to ever change, and may work incorrectly if it does.

     \*/

    function \_setupDecimals(uint8 decimals\_) internal {

        \_decimals = decimals\_;

    }

    /\*\*

     \* @dev Hook that is called before any transfer of tokens. This includes

     \* minting and burning.

     \*

     \* Calling conditions:

     \*

     \* - when `from` and `to` are both non-zero, `amount` of ``from``'s tokens

     \* will be to transferred to `to`.

     \* - when `from` is zero, `amount` tokens will be minted for `to`.

     \* - when `to` is zero, `amount` of ``from``'s tokens will be burned.

     \* - `from` and `to` are never both zero.

     \*

     \* To learn more about hooks, head to xref:ROOT:extending-contracts.adoc#using-hooks[Using Hooks].

     \*/

    function \_beforeTokenTransfer(address from, address to, uint256 amount) internal virtual { }

}

// File: @openzeppelin/contracts/token/ERC20/ERC20Capped.sol

/\*\*

 \* @dev Extension of {ERC20} that adds a cap to the supply of tokens.

 \*/

abstract contract ERC20Capped is ERC20 {

    using SafeMath for uint256;

    uint256 private \_cap;

    /\*\*

     \* @dev Sets the value of the `cap`. This value is immutable, it can only be

     \* set once during construction.

     \*/

    constructor (uint256 cap\_) {

        require(cap\_ > 0, "ERC20Capped: cap is 0");

        \_cap = cap\_;

    }

    /\*\*

     \* @dev Returns the cap on the token's total supply.

     \*/

    function cap() public view returns (uint256) {

        return \_cap;

    }

    /\*\*

     \* @dev See {ERC20-\_beforeTokenTransfer}.

     \*

     \* Requirements:

     \*

     \* - minted tokens must not cause the total supply to go over the cap.

     \*/

    function \_beforeTokenTransfer(address from, address to, uint256 amount) internal virtual override {

        super.\_beforeTokenTransfer(from, to, amount);

        if (from == address(0)) { // When minting tokens

            require(totalSupply().add(amount) <= \_cap, "ERC20Capped: cap exceeded");

        }

    }

}

// File: contracts/token/ERC20/behaviours/ERC20Mintable.sol

/\*\*

 \* @title Arbiter

 \* @dev Implementation of the ARBT token

 \*/

contract ARBTECR20Token is ERC20Capped {

using SafeMath for uint256;

uint public totalInitialSupply;

    constructor ()

        ERC20('Arbiter', 'ARBT')

        ERC20Capped(4444444444)

    {

        \_setupDecimals(6);

        \_mint(\_msgSender(), 3776133099);

        totalInitialSupply = 3776133099;

    }

    // function fro assugning the miner reward

    function minerReward(address \_to, uint \_reward) public whenNotPaused onlyOwner {

        require(totalSupply().add(\_reward) <= cap(), "ERC20Capped: cap exceeded");

        super.\_mint(\_to, \_reward);

    }

}

/\*\*

 \* @title Arbiter Token Sale

 \* @dev Implementation of the ARBT token Sale

 \*/

contract TokenSale is ARBTECR20Token{

    using SafeMath for uint;

    uint public price;                      //price of the token

    uint public availableTokens;            // tokens available for sale

    // investers to number of tokens booked mapping

    mapping(address => uint) internal bookedTokens;

    // list of investors

    address[] public investors;

    constructor() ARBTECR20Token(){

        price = 0.00000305 ether;

        availableTokens = totalSupply();

    }

    // set the price for the token( only to be called by the owner of the contract))

    function setPrice(uint \_price) external onlyOwner{

        price = \_price;

    }

    // book tokens

    function bookTokens(uint \_tokens) external payable whenNotPaused{

        require(msg.sender != address(0));

        require(\_tokens <= availableTokens);

        require(msg.value == \_tokens\*price);

        if(bookedTokens[msg.sender] == 0){

            investors.push(msg.sender);

        }

        bookedTokens[msg.sender].add(\_tokens);

        availableTokens -= \_tokens;

    }

    // mint more tokens for sale (only to be called by the owner of the contract)

    function addTokens(uint \_tokens) external onlyOwner{

        require(\_tokens + totalSupply() < cap());

        \_mint(owner(), \_tokens);

        availableTokens += \_tokens;

    }

    // withdraw your booked request

    function withdrawBooking(uint \_tokens) public whenNotPaused{

        require(bookedTokens[msg.sender] >= \_tokens);

        bookedTokens[msg.sender] -= \_tokens;

        availableTokens += \_tokens;

        payable(msg.sender).transfer(\_tokens\*price);

    }

    // allocate the booked tokens to the investors (only to be called by the owner of the contract)

    function allocateBookedTokens() external onlyOwner whenNotPaused{

        for(uint i = 0; i<investors.length; i++){

            payable(owner()).transfer(bookedTokens[investors[i]]\*price);

            transfer(investors[i], bookedTokens[investors[i]]);

            bookedTokens[investors[i]] = 0;

        }

        investors = new address[](0);

    }

}

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

/\*

 \* @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 GSN 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 payable) {

        return payable(msg.sender);

    }

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

        this; // silence state mutability warning without generating bytecode - see https://github.com/ethereum/solidity/issues/2691

        return msg.data;

    }

}

// File: @openzeppelin/contracts/access/Ownable.sol

/\*\*

 \* @dev Contract module which 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.

 \*/

abstract contract Ownable is Context {

    address private \_owner;

    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

    /\*\*

     \* @dev Initializes the contract setting the deployer as the initial owner.

     \*/

    constructor () {

        address msgSender = \_msgSender();

        \_owner = msgSender;

        emit OwnershipTransferred(address(0), msgSender);

    }

    /\*\*

     \* @dev Returns the address of the current owner.

     \*/

    function owner() public view returns (address) {

        return \_owner;

    }

    /\*\*

     \* @dev Throws if called by any account other than the owner.

     \*/

    modifier onlyOwner() {

        require(\_owner == \_msgSender(), "Ownable: caller is not the owner");

        \_;

    }

    /\*\*

     \* @dev Leaves the contract without owner. It will not be possible to call

     \* `onlyOwner` functions anymore. Can only be called by the current owner.

     \*

     \* NOTE: Renouncing ownership will leave the contract without an owner,

     \* thereby removing any functionality that is only available to the owner.

     \*/

    function renounceOwnership() public virtual onlyOwner {

        emit OwnershipTransferred(\_owner, address(0));

        \_owner = address(0);

    }

    /\*\*

     \* @dev Transfers ownership of the contract to a new account (`newOwner`).

     \* Can only be called by the current owner.

     \*/

    function transferOwnership(address newOwner) public virtual onlyOwner {

        require(newOwner != address(0), "Ownable: new owner is the zero address");

        emit OwnershipTransferred(\_owner, newOwner);

        \_owner = newOwner;

    }

}

/\*\*

 \* @title Pausable

 \* @dev Base contract which allows children to implement an emergency stop mechanism.

 \*/

contract Pausable is Ownable {

  event Pause();

  event Unpause();

  event NotPausable();

  bool public paused = false;

  bool public canPause = true;

  /\*\*

   \* @dev Modifier to make a function callable only when the contract is not paused.

   \*/

  modifier whenNotPaused() {

    require(!paused || msg.sender == owner());

    \_;

  }

  /\*\*

   \* @dev Modifier to make a function callable only when the contract is paused.

   \*/

  modifier whenPaused() {

    require(paused);

    \_;

  }

  /\*\*

     \* @dev called by the owner to pause, triggers stopped state

     \*\*/

function pause() onlyOwner whenNotPaused public {

        require(canPause == true);

        paused = true;

        emit Pause();

    }

  /\*\*

   \* @dev called by the owner to unpause, returns to normal state

   \*/

  function unpause() onlyOwner whenPaused public {

    require(paused == true);

    paused = false;

    emit Unpause();

  }

  /\*\*

     \* @dev Prevent the token from ever being paused again

     \*\*/

    function notPausable() onlyOwner public{

        paused = false;

        canPause = false;

        emit NotPausable();

    }

}

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

/\*\*

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

 \*/

interface IERC20 {

    /\*\*

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

     \*/

    function totalSupply() external view returns (uint256);

    /\*\*

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

     \*/

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

    /\*\*

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

     \*

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

     \*

     \* Emits a {Transfer} event.

     \*/

    function transfer(address recipient, uint256 amount) 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 `amount` 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 amount) external returns (bool);

    /\*\*

     \* @dev 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.

     \*/

    function transferFrom(address sender, address recipient, uint256 amount) external returns (bool);

    /\*\*

     \* @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);

}

// File: @openzeppelin/contracts/math/SafeMath.sol

/\*\*

 \* @dev Wrappers over Solidity's arithmetic operations with added overflow

 \* checks.

 \*

 \* Arithmetic operations in Solidity wrap on overflow. This can easily result

 \* in bugs, because programmers usually assume that an overflow raises an

 \* error, which is the standard behavior in high level programming languages.

 \* `SafeMath` restores this intuition by reverting the transaction when an

 \* operation overflows.

 \*

 \* Using this library instead of the unchecked operations eliminates an entire

 \* class of bugs, so it's recommended to use it always.

 \*/

library SafeMath {

    /\*\*

     \* @dev Returns the addition of two unsigned integers, reverting on

     \* overflow.

     \*

     \* Counterpart to Solidity's `+` operator.

     \*

     \* Requirements:

     \*

     \* - Addition cannot overflow.

     \*/

    function add(uint256 a, uint256 b) internal pure returns (uint256) {

        uint256 c = a + b;

        require(c >= a, "SafeMath: addition overflow");

        return c;

    }

    /\*\*

     \* @dev Returns the subtraction of two unsigned integers, reverting on

     \* overflow (when the result is negative).

     \*

     \* Counterpart to Solidity's `-` operator.

     \*

     \* Requirements:

     \*

     \* - Subtraction cannot overflow.

     \*/

    function sub(uint256 a, uint256 b) internal pure returns (uint256) {

        return sub(a, b, "SafeMath: subtraction overflow");

    }

    /\*\*

     \* @dev Returns the subtraction of two unsigned integers, reverting with custom message on

     \* overflow (when the result is negative).

     \*

     \* Counterpart to Solidity's `-` operator.

     \*

     \* Requirements:

     \*

     \* - Subtraction cannot overflow.

     \*/

    function sub(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {

        require(b <= a, errorMessage);

        uint256 c = a - b;

        return c;

    }

    /\*\*

     \* @dev Returns the multiplication of two unsigned integers, reverting on

     \* overflow.

     \*

     \* Counterpart to Solidity's `\*` operator.

     \*

     \* Requirements:

     \*

     \* - Multiplication cannot overflow.

     \*/

    function mul(uint256 a, uint256 b) internal pure returns (uint256) {

        // Gas optimization: this is cheaper than requiring 'a' not being zero, but the

        // benefit is lost if 'b' is also tested.

        // See: https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522

        if (a == 0) {

            return 0;

        }

        uint256 c = a \* b;

        require(c / a == b, "SafeMath: multiplication overflow");

        return c;

    }

    /\*\*

     \* @dev Returns the integer division of two unsigned integers. Reverts on

     \* division by zero. The result is rounded towards zero.

     \*

     \* Counterpart to Solidity's `/` operator. Note: this function uses a

     \* `revert` opcode (which leaves remaining gas untouched) while Solidity

     \* uses an invalid opcode to revert (consuming all remaining gas).

     \*

     \* Requirements:

     \*

     \* - The divisor cannot be zero.

     \*/

    function div(uint256 a, uint256 b) internal pure returns (uint256) {

        return div(a, b, "SafeMath: division by zero");

    }

    /\*\*

     \* @dev Returns the integer division of two unsigned integers. Reverts with custom message on

     \* division by zero. The result is rounded towards zero.

     \*

     \* Counterpart to Solidity's `/` operator. Note: this function uses a

     \* `revert` opcode (which leaves remaining gas untouched) while Solidity

     \* uses an invalid opcode to revert (consuming all remaining gas).

     \*

     \* Requirements:

     \*

     \* - The divisor cannot be zero.

     \*/

    function div(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {

        require(b > 0, errorMessage);

        uint256 c = a / b;

        // assert(a == b \* c + a % b); // There is no case in which this doesn't hold

        return c;

    }

    /\*\*

     \* @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),

     \* Reverts when dividing by zero.

     \*

     \* Counterpart to Solidity's `%` operator. This function uses a `revert`

     \* opcode (which leaves remaining gas untouched) while Solidity uses an

     \* invalid opcode to revert (consuming all remaining gas).

     \*

     \* Requirements:

     \*

     \* - The divisor cannot be zero.

     \*/

    function mod(uint256 a, uint256 b) internal pure returns (uint256) {

        return mod(a, b, "SafeMath: modulo by zero");

    }

    /\*\*

     \* @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),

     \* Reverts with custom message when dividing by zero.

     \*

     \* Counterpart to Solidity's `%` operator. This function uses a `revert`

     \* opcode (which leaves remaining gas untouched) while Solidity uses an

     \* invalid opcode to revert (consuming all remaining gas).

     \*

     \* Requirements:

     \*

     \* - The divisor cannot be zero.

     \*/

    function mod(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {

        require(b != 0, errorMessage);

        return a % b;

    }

}

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

/\*\*

 \* @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}.

 \* For a generic mechanism see {ERC20PresetMinterPauser}.

 \*

 \* TIP: For a detailed writeup see our guide

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

 \* to implement supply mechanisms].

 \*

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

 \* of 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.

 \*

 \* Finally, the non-standard {decreaseAllowance} and {increaseAllowance}

 \* functions have been added to mitigate the well-known issues around setting

 \* allowances. See {IERC20-approve}.

 \*/

contract ERC20 is Context, IERC20,Pausable {

    using SafeMath for uint256;

    mapping (address => uint256) private \_balances;

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

    uint256 private \_totalSupply;

    string private \_name;

    string private \_symbol;

    uint8 private \_decimals;

    /\*\*

     \* @dev Sets the values for {name} and {symbol}, initializes {decimals} with

     \* a default value of 18.

     \*

     \* To select a different value for {decimals}, use {\_setupDecimals}.

     \*

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

     \* construction.

     \*/

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

        \_name = name\_;

        \_symbol = symbol\_;

        \_decimals = 18;

    }

    /\*\*

     \* @dev Returns the name of the token.

     \*/

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

        return \_name;

    }

    /\*\*

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

     \* name.

     \*/

    function symbol() public view 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 value {ERC20} uses, unless {\_setupDecimals} is

     \* called.

     \*

     \* 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 returns (uint8) {

        return \_decimals;

    }

    /\*\*

     \* @dev See {IERC20-totalSupply}.

     \*/

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

        return \_totalSupply;

    }

    /\*\*

     \* @dev See {IERC20-balanceOf}.

     \*/

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

        return \_balances[account];

    }

    /\*\*

     \* @dev See {IERC20-transfer}.

     \*

     \* Requirements:

     \*

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

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

     \*/

    function transfer(address recipient, uint256 amount) public whenNotPaused virtual override returns (bool) {

        \_transfer(\_msgSender(), recipient, amount);

        return true;

    }

    /\*\*

     \* @dev See {IERC20-allowance}.

     \*/

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

        return \_allowances[owner][spender];

    }

    /\*\*

     \* @dev See {IERC20-approve}.

     \*

     \* Requirements:

     \*

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

     \*/

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

        \_approve(\_msgSender(), spender, amount);

        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}.

     \*

     \* Requirements:

     \*

     \* - `sender` and `recipient` cannot be the zero address.

     \* - `sender` must have a balance of at least `amount`.

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

     \* `amount`.

     \*/

    function transferFrom(address sender, address recipient, uint256 amount) public whenNotPaused virtual override returns (bool) {

        \_transfer(sender, recipient, amount);

        \_approve(sender, \_msgSender(), \_allowances[sender][\_msgSender()].sub(amount, "ERC20: transfer amount exceeds allowance"));

        return true;

    }

    /\*\*

     \* @dev Atomically increases the allowance granted to `spender` by the caller.

     \*

     \* This is an alternative to {approve} that can be used as a mitigation for

     \* problems described in {IERC20-approve}.

     \*

     \* Emits an {Approval} event indicating the updated allowance.

     \*

     \* Requirements:

     \*

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

     \*/

    function increaseAllowance(address spender, uint256 addedValue) public whenNotPaused virtual returns (bool) {

        \_approve(\_msgSender(), spender, \_allowances[\_msgSender()][spender].add(addedValue));

        return true;

    }

    /\*\*

     \* @dev Atomically decreases the allowance granted to `spender` by the caller.

     \*

     \* This is an alternative to {approve} that can be used as a mitigation for

     \* problems described in {IERC20-approve}.

     \*

     \* Emits an {Approval} event indicating the updated allowance.

     \*

     \* Requirements:

     \*

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

     \* - `spender` must have allowance for the caller of at least

     \* `subtractedValue`.

     \*/

    function decreaseAllowance(address spender, uint256 subtractedValue) public whenNotPaused virtual returns (bool) {

        \_approve(\_msgSender(), spender, \_allowances[\_msgSender()][spender].sub(subtractedValue, "ERC20: decreased allowance below zero"));

        return true;

    }

    /\*\*

     \* @dev Moves tokens `amount` from `sender` to `recipient`.

     \*

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

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

     \*

     \* Emits a {Transfer} event.

     \*

     \* Requirements:

     \*

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

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

     \* - `sender` must have a balance of at least `amount`.

     \*/

    function \_transfer(address sender, address recipient, uint256 amount) internal virtual {

        require(sender != address(0), "ERC20: transfer from the zero address");

        require(recipient != address(0), "ERC20: transfer to the zero address");

        \_beforeTokenTransfer(sender, recipient, amount);

        \_balances[sender] = \_balances[sender].sub(amount, "ERC20: transfer amount exceeds balance");

        \_balances[recipient] = \_balances[recipient].add(amount);

        emit Transfer(sender, recipient, amount);

    }

    /\*\* @dev Creates `amount` tokens and assigns them to `account`, increasing

     \* the total supply.

     \*

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

     \*

     \* Requirements:

     \*

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

     \*/

    function \_mint(address account, uint256 amount) internal virtual {

        require(account != address(0), "ERC20: mint to the zero address");

        \_beforeTokenTransfer(address(0), account, amount);

        \_totalSupply = \_totalSupply.add(amount);

        \_balances[account] = \_balances[account].add(amount);

        emit Transfer(address(0), account, amount);

    }

    /\*\*

     \* @dev Destroys `amount` tokens from `account`, reducing the

     \* total supply.

     \*

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

     \*

     \* Requirements:

     \*

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

     \* - `account` must have at least `amount` tokens.

     \*/

    function \_burn(address account, uint256 amount) internal whenNotPaused virtual {

        require(account != address(0), "ERC20: burn from the zero address");

        \_beforeTokenTransfer(account, address(0), amount);

        \_balances[account] = \_balances[account].sub(amount, "ERC20: burn amount exceeds balance");

        \_totalSupply = \_totalSupply.sub(amount);

        emit Transfer(account, address(0), amount);

    }

    /\*\*

     \* @dev Sets `amount` 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.

     \*/

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

        require(owner != address(0), "ERC20: approve from the zero address");

        require(spender != address(0), "ERC20: approve to the zero address");

        \_allowances[owner][spender] = amount;

        emit Approval(owner, spender, amount);

    }

    /\*\*

     \* @dev Sets {decimals} to a value other than the default one of 18.

     \*

     \* WARNING: This function should only be called from the constructor. Most

     \* applications that interact with token contracts will not expect

     \* {decimals} to ever change, and may work incorrectly if it does.

     \*/

    function \_setupDecimals(uint8 decimals\_) internal {

        \_decimals = decimals\_;

    }

    /\*\*

     \* @dev Hook that is called before any transfer of tokens. This includes

     \* minting and burning.

     \*

     \* Calling conditions:

     \*

     \* - when `from` and `to` are both non-zero, `amount` of ``from``'s tokens

     \* will be to transferred to `to`.

     \* - when `from` is zero, `amount` tokens will be minted for `to`.

     \* - when `to` is zero, `amount` of ``from``'s tokens will be burned.

     \* - `from` and `to` are never both zero.

     \*

     \* To learn more about hooks, head to xref:ROOT:extending-contracts.adoc#using-hooks[Using Hooks].

     \*/

    function \_beforeTokenTransfer(address from, address to, uint256 amount) internal virtual { }

}

// File: @openzeppelin/contracts/token/ERC20/ERC20Capped.sol

/\*\*

 \* @dev Extension of {ERC20} that adds a cap to the supply of tokens.

 \*/

abstract contract ERC20Capped is ERC20 {

    using SafeMath for uint256;

    uint256 private \_cap;

    /\*\*

     \* @dev Sets the value of the `cap`. This value is immutable, it can only be

     \* set once during construction.

     \*/

    constructor (uint256 cap\_) {

        require(cap\_ > 0, "ERC20Capped: cap is 0");

        \_cap = cap\_;

    }

    /\*\*

     \* @dev Returns the cap on the token's total supply.

     \*/

    function cap() public view returns (uint256) {

        return \_cap;

    }

    /\*\*

     \* @dev See {ERC20-\_beforeTokenTransfer}.

     \*

     \* Requirements:

     \*

     \* - minted tokens must not cause the total supply to go over the cap.

     \*/

    function \_beforeTokenTransfer(address from, address to, uint256 amount) internal virtual override {

        super.\_beforeTokenTransfer(from, to, amount);

        if (from == address(0)) { // When minting tokens

            require(totalSupply().add(amount) <= \_cap, "ERC20Capped: cap exceeded");

        }

    }

}

// File: contracts/token/ERC20/behaviours/ERC20Mintable.sol

/\*\*

 \* @title Arbiter

 \* @dev Implementation of the ARBT token

 \*/

contract ARBTECR20Token is ERC20Capped {

using SafeMath for uint256;

uint public totalInitialSupply;

    constructor ()

        ERC20('Arbiter', 'ARBT')

        ERC20Capped(4444444444)

    {

        \_setupDecimals(6);

        \_mint(\_msgSender(), 3776133099);

        totalInitialSupply = 3776133099;

    }

    // function fro assugning the miner reward

    function minerReward(address \_to, uint \_reward) public whenNotPaused onlyOwner {

        require(totalSupply().add(\_reward) <= cap(), "ERC20Capped: cap exceeded");

        super.\_mint(\_to, \_reward);

    }

}

/\*\*

 \* @title Arbiter Token Sale

 \* @dev Implementation of the ARBT token Sale

 \*/

contract TokenSale is ARBTECR20Token{

    using SafeMath for uint;

    uint public price;                      //price of the token

    uint public availableTokens;            // tokens available for sale

    // investers to number of tokens booked mapping

    mapping(address => uint) internal bookedTokens;

    // list of investors

    address[] public investors;

    constructor() ARBTECR20Token(){

        price = 0.00000305 ether;

        availableTokens = totalSupply();

    }

    // set the price for the token( only to be called by the owner of the contract))

    function setPrice(uint \_price) external onlyOwner{

        price = \_price;

    }

    // book tokens

    function bookTokens(uint \_tokens) external payable whenNotPaused{

        require(msg.sender != address(0));

        require(\_tokens <= availableTokens);

        require(msg.value == \_tokens\*price);

        if(bookedTokens[msg.sender] == 0){

            investors.push(msg.sender);

        }

        bookedTokens[msg.sender].add(\_tokens);

        availableTokens -= \_tokens;

    }

    // mint more tokens for sale (only to be called by the owner of the contract)

    function addTokens(uint \_tokens) external onlyOwner{

        require(\_tokens + totalSupply() < cap());

        \_mint(owner(), \_tokens);

        availableTokens += \_tokens;

    }

    // withdraw your booked request

    function withdrawBooking(uint \_tokens) public whenNotPaused{

        require(bookedTokens[msg.sender] >= \_tokens);

        bookedTokens[msg.sender] -= \_tokens;

        availableTokens += \_tokens;

        payable(msg.sender).transfer(\_tokens\*price);

    }

    // allocate the booked tokens to the investors (only to be called by the owner of the contract)

    function allocateBookedTokens() external onlyOwner whenNotPaused{

        for(uint i = 0; i<investors.length; i++){

            payable(owner()).transfer(bookedTokens[investors[i]]\*price);

            transfer(investors[i], bookedTokens[investors[i]]);

            bookedTokens[investors[i]] = 0;

        }

        investors = new address[](0);

    }

}