```
// SPDX-License-Identifier: MIT
```

This is a license identifier. It tells others that you're using the open-source MIT license for your contract. It's required for code transparency and legal clarity.

```
pragma solidity >=0.5.0 <0.8.0;
```
This program works with version 5, 6, and 7 of the language, but not 8.

```
contract Factory {
```
This line starts your **smart contract**, like creating a new class in other languages. You're naming your contract `Factory`.

```
    uint idDigits = 16;
```
This declares a variable named `idDigits` of type `uint` (which stands for "unsigned integer" — only positive numbers).
You're setting it to 16. This means you want your product ID to be 16 digits long.

```
    uint idModulus = 10 ** idDigits;
```
This line means: `10 to the power of 16`, which equals 10,000,000,000,000,000 — a number with 16 digits.
We use this to "cut down" any big number into exactly 16 digits using the modulus (%) operator.

```
    struct Product { //structure of our product organised by name and id
    string name;
    uint id;
```
A `struct` is like a custom data type.
You're defining a `Product` type that has:

- a `name` (text),
- an `id` (number).

So you can store multiple attributes in one thing.

```
}
```

```
Product[] public products; // create array in dynamic mode
```
This creates a **public array (list)** to store all your products.
Anyone can access this list, and each item in it is of type `Product`.

```
event NewProduct(uint ArrayProductId, string name, uint id);
```
This defines an **event**.
Events are like messages sent to your app (front-end) whenever something happens.
Here, when a new product is added, the blockchain will broadcast:

- the product's index in the array,
- its name,
- its ID.

```
//Ownership mappings
mapping(uint => address) public productToOwner;
```
A `mapping` is like a dictionary or hashmap.
Here, you're saying: for each product ID (uint), store **which address** (wallet) owns it.

```
mapping(address => uint) ownerProductCount;
```
Another mapping, but this one counts **how many products** each wallet address owns.

```
function _createProduct(string memory _name, uint _id) private { // this function is gonna add new product id and name to our array
```
This is a **private function** that:

- takes a name (text) and ID (number),
- creates a product, and
- adds it to your system.
  Only the contract itself can call this.

```
products.push(Product(_name, _id)); // this line gonna add the name and id by using push command

uint productId = products.length - 1;
products.push(Product(_name, _id));
```

You're creating a new product and pushing it to the array. `push()` adds it to the end and returns the new length of the array. You subtract 1 to get its **index** (because arrays start from 0).

```
(       uint productId = products.push(Product(_name, _id)) - 1;)
```

```
// Track ownership
productToOwner[productId] = msg.sender;
```
You store **who created the product** (msg.sender = person who called the function).
```
ownerProductCount[msg.sender]++;
```
You add +1 to the product count for that wallet.

```
emit NewProduct(productId, _name, _id);
```
This **triggers the event**, sending info to the outside world that a new product was created.

```
}


function _generateRandomId(string memory _str) private view returns (uint) { //view functions can read the blockchain's state (like variables or mappings) but cannot modify it

A private view function — it doesn't change any data, only looks at it.
Takes a string and returns a number.
This is how we create a random-looking product ID.

uint rand = uint(keccak256(abi.encodePacked(_str))); //Application Binary Interface.
This uses a hash function (keccak256) to convert your text into a big number.
abi.encodePacked packs the string into binary first.
The result is converted into a uint.

return rand % idModulus;

This cuts down the big hash to a 16-digit number using %.
Example: 235235253426252345 % 10000000000000000 will give you a 16-digit result.
}

function createRandomProduct(string memory _name) public {
A public function — anyone can call it.
Takes a name, generates a random ID, and creates a product.

uint randId = _generateRandomId(_name);
You create a random ID using the name input.

_createProduct(_name, randId);
Then you call the private function to actually create and store the product.
}

function Ownership(uint _productId) public {

productToOwner[_productId] = msg.sender;
ownerProductCount[msg.sender]++;
}


function getProductsByOwner(address _owner) external view returns (uint[] memory) {
This is a function to get all products owned by an address.
external means only called from outside.
view means it doesn't change anything.
It returns a list of product IDs.

uint counter = 0;
You start a counter to keep track of results.
```

```
uint[] memory result = new uint[](ownerProductCount[_owner]);
Create an array of the right size to hold all the owner's products.

for (uint i = 0; i < products.length; i++) {
Loop through all products...

if (productToOwner[i] == _owner) {
...and check if the product at index i is owned by _owner.

result[counter] = i;

If yes, save its index in the result array.

counter++;
}
}

return result;

Return the array of product indices owned by _owner.
}


}

===// contracts/access-control/Auth.sol

// SPDX-License-Identifier: MIT
pragma solidity >=0.5.0 <0.8.0; //Only compile this contract with Solidity version 0.5.0 or newer, but not 0.8.0 or above."

contract Auth {
Here, we define a new contract named Auth.
```

- `contract` is the keyword used to define a smart contract in Solidity.
- `Auth` is the name of the contract. You can think of it as a class or module in traditional programming.

```
address private _administrator;
```

`address` is a **data type** in Solidity used to store Ethereum addresses (which are 160-bit values).

- `_administrator` is a **state variable** that holds the address of the administrator of the contract.
- `private` means this variable can only be accessed inside this contract and is not accessible from outside the contract.

```
constructor(address deployer) {
// Make the deployer of the contract the administrator
_administrator = deployer;
}
```

The `constructor` is a special function that **runs only once** when the contract is deployed. It's used to initialize the contract's state.

- `address deployer` means the constructor expects an **address** to be passed in when the contract is deployed. This will be the Ethereum address of the contract's deployer (the person or account creating the contract).

In this line, the `_administrator` state variable is assigned the **address** passed into the constructor.

- `deployer` is the argument passed when deploying the contract.
- This makes the deployer's address the **administrator** of the contract, meaning they will have special permissions or control over it.

```
function isAdministrator(address user) public view returns (bool) {
```

This line defines a **public function** named `isAdministrator`.

- `public` means this function can be called by anyone (outside or inside the contract).
- `view` means the function does not modify the contract's state, only reads from it.
- `returns (bool)` means the function will return a **boolean value** (`true` or `false`).

This function checks if a given address (`user`) is the administrator of the contract.

```
return user == _administrator;
```

This line returns **true** or **false** based on whether the passed address (`user`) is equal to the stored `_administrator` address.

- If `user` is the same as `_administrator`, the function returns `true`.
- If `user` is different from `_administrator`, the function returns `false`.

This provides a way for other contracts or users to verify if a certain address is the administrator.

```
}
}
```

Recap:

- **This contract** is called `Auth` and is used to **store the address** of the contract's administrator.
- The contract's deployer becomes the **administrator**.
- The function `isAdministrator` allows anyone to check if an address is the administrator or not.

==========

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.5.0 <0.8.0; //Only compile this contract with Solidity version 0.5.0 or newer, but not 0.8.0 or above."

contract UserValidation {
// Public mapping from uint to uint to store user's age
```

This line starts the definition of the contract.

- `contract` is the keyword that defines a new smart contract in Solidity.
- `UserValidation` is the name of the contract. It is used to perform user validation based on their age.

```
mapping(uint => uint) public age;
```

Here, a **mapping** is defined:

- `mapping` is a key-value store, similar to a dictionary in Python or a hash map in other programming languages.
- `uint => uint` means the key is an unsigned integer (`uint`), and the value is also an unsigned integer (`uint`).
- `age` is the name of the mapping, and it will store **user IDs as keys** and their corresponding **ages as values**.

Because `age` is declared `public`, Solidity will automatically generate a getter function for it, allowing other contracts and users to retrieve the age for a given `userId` by calling `age(userId)`.

```
// Modifier to check if user is older than a certain age
modifier olderThan(uint _age, uint _userId) {
```

`modifier` is a special type of function that can modify the behavior of other functions. Modifiers are often used for conditions that must be checked before the function is executed.

- `uint _age` is a parameter that specifies the age threshold for comparison.
- `uint _userId` is the parameter representing the user ID whose age needs to be checked.

```
require(age[_userId] >= _age, "User is not old enough");
```
This line checks if the user is old enough:

- `age[_userId]` retrieves the user's age from the `age` mapping using their `userId`.
- `>= _age` checks if the user's age is **greater than or equal to** the `_age` parameter passed into the modifier.
- If the condition is **not met**, the `require` statement will **throw an error** and revert the transaction, displaying the error message `"User is not old enough"`.

The `require` statement ensures that the age check passes before the function can execute further.

```
_;
```
This is a placeholder that is used to indicate where the modified function's code should be inserted.

- The `_;` represents the rest of the code in the function that uses the modifier.
- After the `require` statement in the modifier checks the condition, Solidity will continue executing the rest of the function (the code after `_;`) if the condition passes.

```
}

// Function that uses the modifier to check if the user is an adult (18+)
function validationUsers(uint _userId) public view olderThan(18, _userId) {
// Function body intentionally left blank
```

This is the function `validationUsers`:

- `validationUsers(uint _userId)` takes one argument: a `userId` (the unique identifier of the user).
- `public` means the function can be called by anyone (both externally and internally).
- `view` means the function only reads the state and does not modify the state.
- `olderThan(18, _userId)` is the **modifier** applied to this function. It checks whether the user with the specified `userId` is at least 18 years old before the function executes.

```
}
}
```

```
// SPDX-License-Identifier: MIT

pragma solidity >=0.5.0 <0.8.0;

contract DemoContract {
// Owner of the contract
address public owner;
```

State Variable `owner`: The `address` type stores Ethereum addresses. This `owner` variable is public, meaning that other contracts and external entities can view it. It holds the address of the contract owner.

- **public** keyword automatically creates a getter function for this variable.

```
// Struct to hold user data
struct Receivers {
string name;
uint256 tokens;
```

Struct Declaration: A struct in Solidity is a user-defined data structure that groups together multiple variables of different types. In this case, the `Receivers` struct holds information about a receiver:

- `name`: A string representing the name of the receiver.
- `tokens`: A `uint256` (unsigned integer of 256 bits) to represent the number of tokens that the receiver has.

```
}

// Mapping to associate an address with a Receiver
mapping(address => Receivers) public users;
```

Mapping Declaration: This declares a `public` mapping named `users`, which maps an `address` to a `Receivers` struct. It keeps track of each user's information, including their name and tokens.

- **Mapping**: A mapping in Solidity is like a hash table or dictionary. Here, the key is an Ethereum address (`address`), and the value is the `Receivers` struct that holds the user's name and token balance.

```
// Modifier to restrict function access to only the contract owner
modifier onlyOwner() {
require(msg.sender == owner, "Not the contract owner");
```

Modifier Declaration (`onlyOwner`): A modifier is a reusable piece of code that can be applied to functions to add additional checks or behavior.

- **`require(msg.sender == owner, "Not the contract owner")`**: This checks whether the function caller (`msg.sender`) is the same as the `owner`. If the caller is not the owner, it reverts the transaction with the error message `"Not the contract owner"`.
- **`_;`**: This is a placeholder where the modified function's code will be inserted. The function will run after the modifier's code is executed.

```
_;
}

// Constructor sets the deployer as owner and gives them 100 tokens
constructor() {
owner = msg.sender;
users[owner].tokens = 100;
```

Constructor: A constructor is a special function that is executed once when the contract is deployed. It is used to initialize the contract state.

- **`owner = msg.sender;`**: The `msg.sender` is the address that deploys the contract. This line sets the `owner` to the address that deploys the contract.

```
}

// Pure function to double an input value
function double(uint _value) public pure returns (uint) {
return _value * 2;
}

// Register the sender with a name
function register(string memory _name) public {
users[msg.sender].name = _name;
}

// Transfer tokens from the owner to another address
function giveToken(address _receiver, uint256 _amount) public onlyOwner {
require(users[owner].tokens >= _amount, "Not enough tokens");
users[owner].tokens -= _amount;
users[_receiver].tokens += _amount;
}
}
```

Function `giveToken`: This function allows the owner to transfer tokens from their balance to another user.

- **Function signature**: `giveToken(address _receiver, uint256 _amount)`: This function takes two arguments:
  - `_receiver`: The address of the user receiving the tokens.
  - `_amount`: The number of tokens to be transferred.

- **`public`**: This allows anyone to call the function, but the modifier `onlyOwner` ensures that only the owner can execute it.
- **`onlyOwner` modifier**: This modifier ensures that only the owner can call the function.
- **`require(users[owner].tokens >= _amount, "Not enough tokens")`**: This checks whether the owner has enough tokens to transfer the specified amount. If the owner does not have enough tokens, the function reverts with the error message `"Not enough tokens"`.
- **`users[owner].tokens -= _amount;`**: This reduces the owner's token balance by the specified `_amount`.
- **`users[_receiver].tokens += _amount;`**: This increases the receiver's token balance by the specified `_amount`.