Solidity **data Types**:

1. **Value Type**
2. **Reference Type**

**Value Type**

Value Type ⇒ holds the data(value) directly within the memory owned by it

These Types stored the values with them instead of elsewhere

Value Type ⇒ do not take more than 32 bytes of memory in size (maximum size limit in solidity )

**Reference Type**:

⇒ store the reference of value stored variable

⇒ can take ake more than 32 bytes of memory in size (maximum size limit in solidity )

⇒ Solidity provides the following Reference Type

1. **Arrays**
2. **Struct**
3. **String**
4. **Mapping**

⇒ In Reference copy of the pointer is assigned not the values

**Rules For Data Location:**

**Rule 1:**

Variable declared as state-level outside the functions and at contract level declared as a Storage variable(Global variable)

⇒ you do not need to write a storage keyword(its automatically declared)

⇒ state variable may be a value type or reference type

**Rule 2:**

The default location of variables created in the function parameter(maybe int, bool) is **memory**

If you want to create a Reference type variable in parameter (like string or array) then you need to write explicitly **memory** keyword otherwise it will not compile

⇒ **mapping** can not be defined in memory level

**Rule 3:**

⇒ Rule 3 is basically for variables inside the function body and contain 5 additional case

**Case 1 ⇒** Variables declared inside the function if value type (int, bool, unit, enum) are stored in memory level

**Case 2** ⇒ For reference variables its compulsory to mention either its memory level or storage level

reference variables declared storage type inside the function can not store the direct value they always point state variable

**Case 3 ⇒** If its memory level e.g. single or double quotes allowed

String memory a = ‘Hello’

Compile

**Case 4 ⇒** If its storage level e.g only double quotes allowed

String storage a = ‘Hello’

Error

**Case 5 ⇒**  Mapping will always be a storage type variable. They can not be declared inside the function or as memory type

⇒ It’s compulsory to tell the variable inside the function body is memory or storage type

**Rule 4:**

⇒ The argument in function calls will be stored in calldata location (which is immutable)

**Rule 5**

**Rule 5 ⇒**  when value type or reference type variable is assigned by another value or reference type then it will be **Pass By Copy** not Pass by reference.

The change will not affect by other variables

e.g. int[2] age = [int(20),30];

int[2] age2 = [int(50),60];

Function testAge() public returns(int) {

age = age2;

age2[0] = 100;

return age[0]; // it will return 50 not the 100 because it is passed by copy;

}

**Rule 6:**

**⇒ memory = Storage**

Assignment of value type from memory to state also create a copy(Its passed by Copy)

e.g. int age = 30;

int age2 = 50;

function testAge() public returns(int) {

age = age2;

age2 = 100;

return age[0]; // it will return 50 not the 100 because it is passed by copy;

}

**Rule 7:**

**⇒ Storage = memory**

**⇒** It will also be **Pass by copy** either its value type or Reference type(changes will not affect to copy variables)

**Rule 8:**

**Value type** variable will be **passed by value** in another memory type variable

function testAge() public view returns(int) {

int age = 30;

int age2 = 15;

age = age2;

age2 = 100;

return age; // it will return 15 not the 100 because it is passed by copy;

}

}

**Reference Type** variable will be **passed by Reference** in another memory type variable

function testAge() public view returns(int ) {

int[2] memory age = [int(20),30];

int[2] memory age2 = [int(50),60];

age = age2;

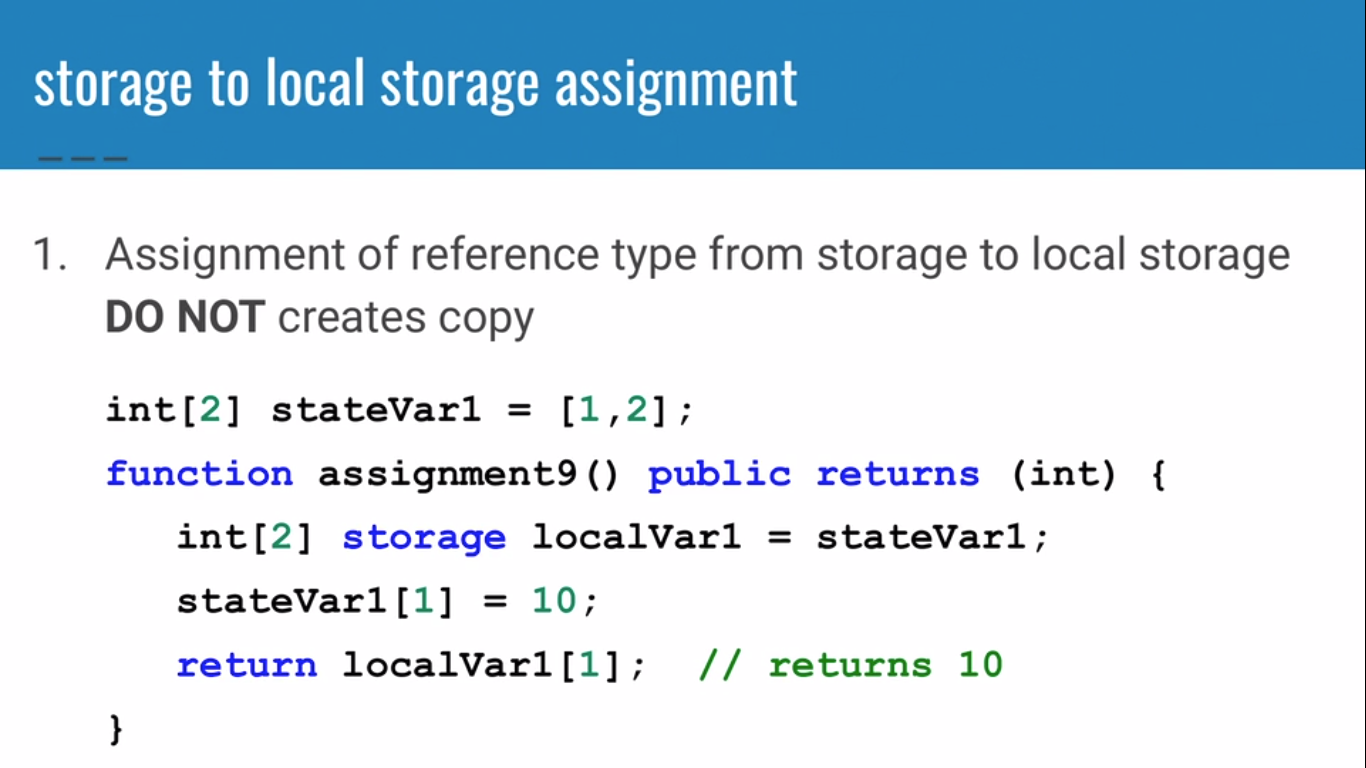
age2[0] = 300;

return age[0]; // it will return 300 the change value because its passed by refernce;

}

**Storage to local Storage**

**Storage = storage ⇒** it ispassed by Reference



**Literals:**

⇒ literals are the **hard coded** value which remain constant e.g int a = 5;

5 is the constant value

**Integeres:**

⇒ Two types of Integers

1. Signed
2. Unsigned

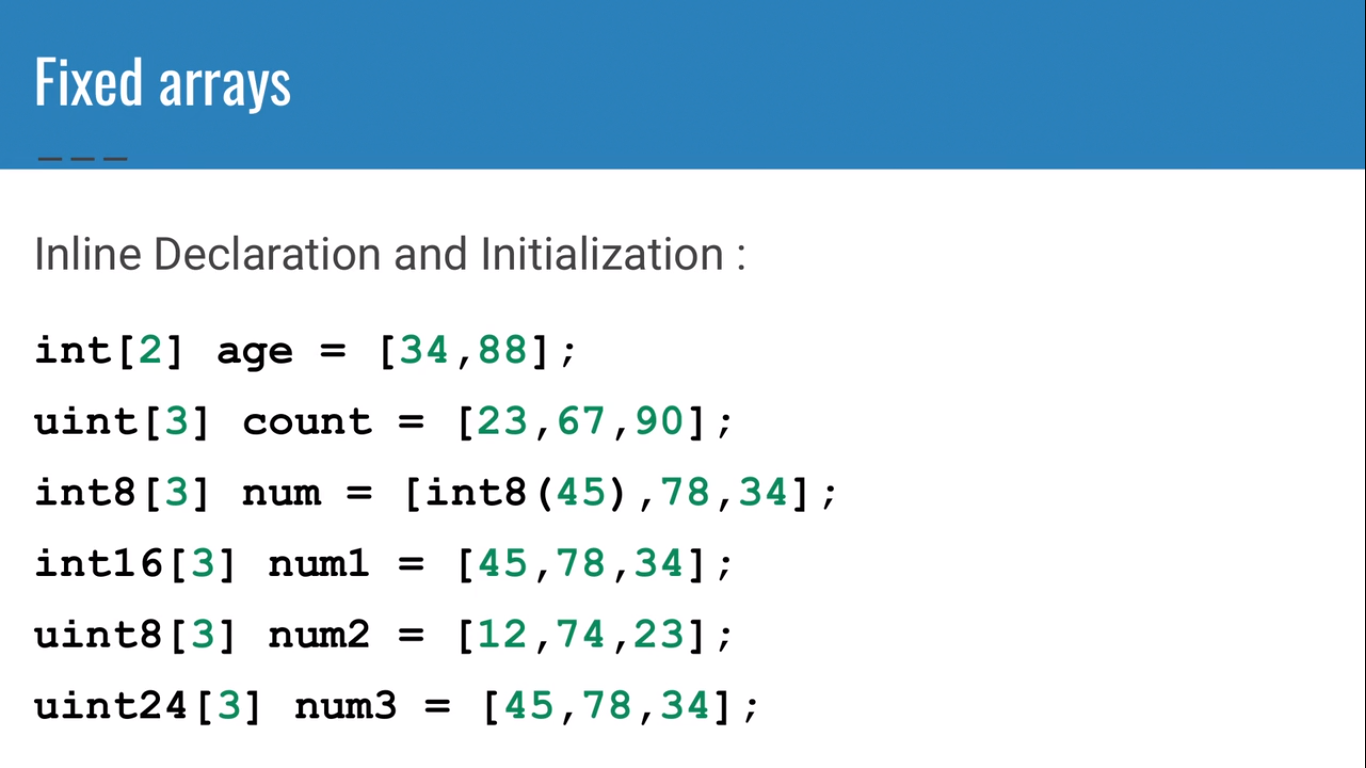
**Bool:**

⇒ In solidity bool can not be converted into inetegeres like Javascript

**Array:**

1. Fixed Array
2. Dynamic Array

⇒ fixed array



**Array will return default value 0 if its empty**

int[4] list;

function testAge() public returns(int) {

return list[0];

}

}

**Dynamic Array:**

⇒ Two ways to declare

int[] list = [1,2,3];

int[] list = new int[] (5);

⇒ lenght can be modified in only **Dynamic array** (not in fixed array)

int[] list = [1,2,3];

// lenght is 3

List.length = 10;

//lenght is 10

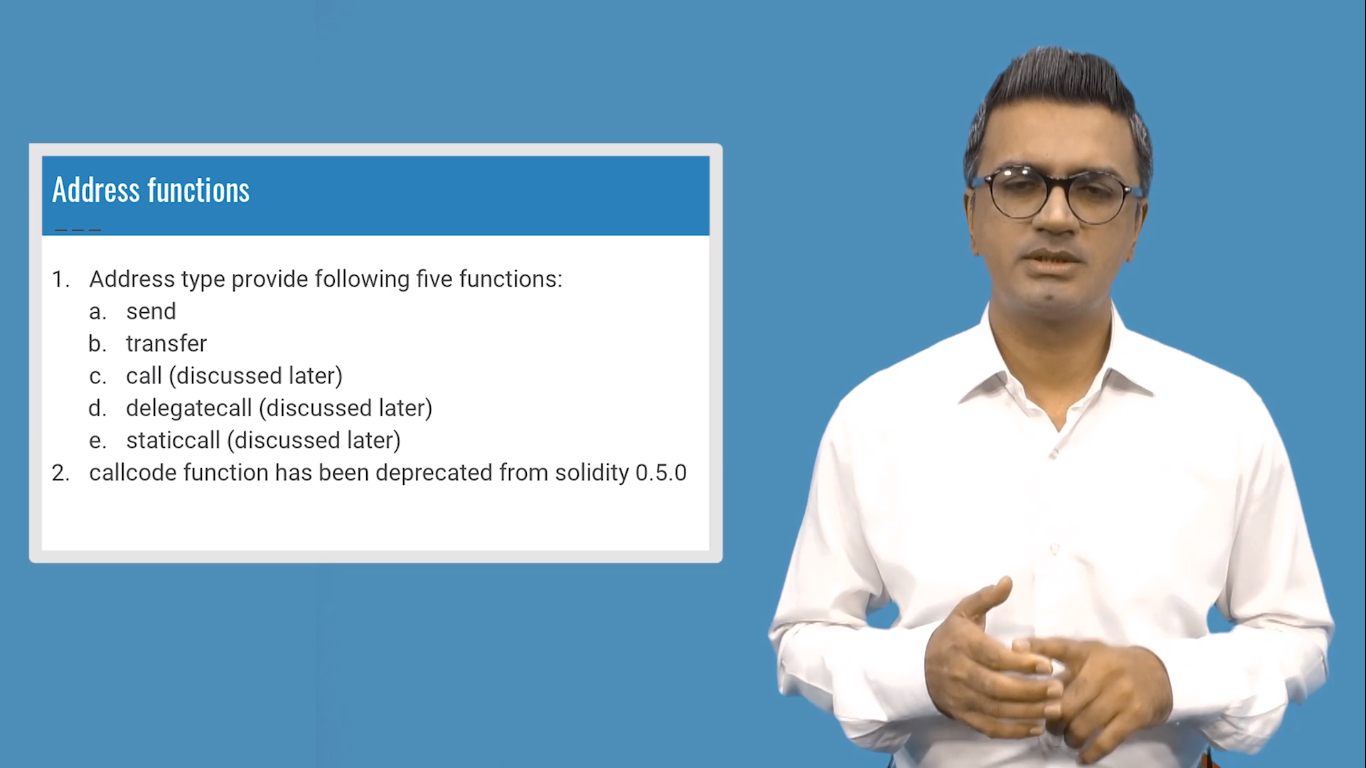
**Solidity Special Array:**

1. Bytes Array ⇒ store raw data in binary format
2. String Array

**Address Data Type:**

⇒ address is a 20 bytes data type which is used to hold the account address in ethereum which are 160 bits or 20 bytes in size.

**Address Payable ⇒**  used to send or receive balance(ether) while only **address can not send ether**



**Sending Ether**

contract MyBalance {

address payable myAddress2 = 0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db;

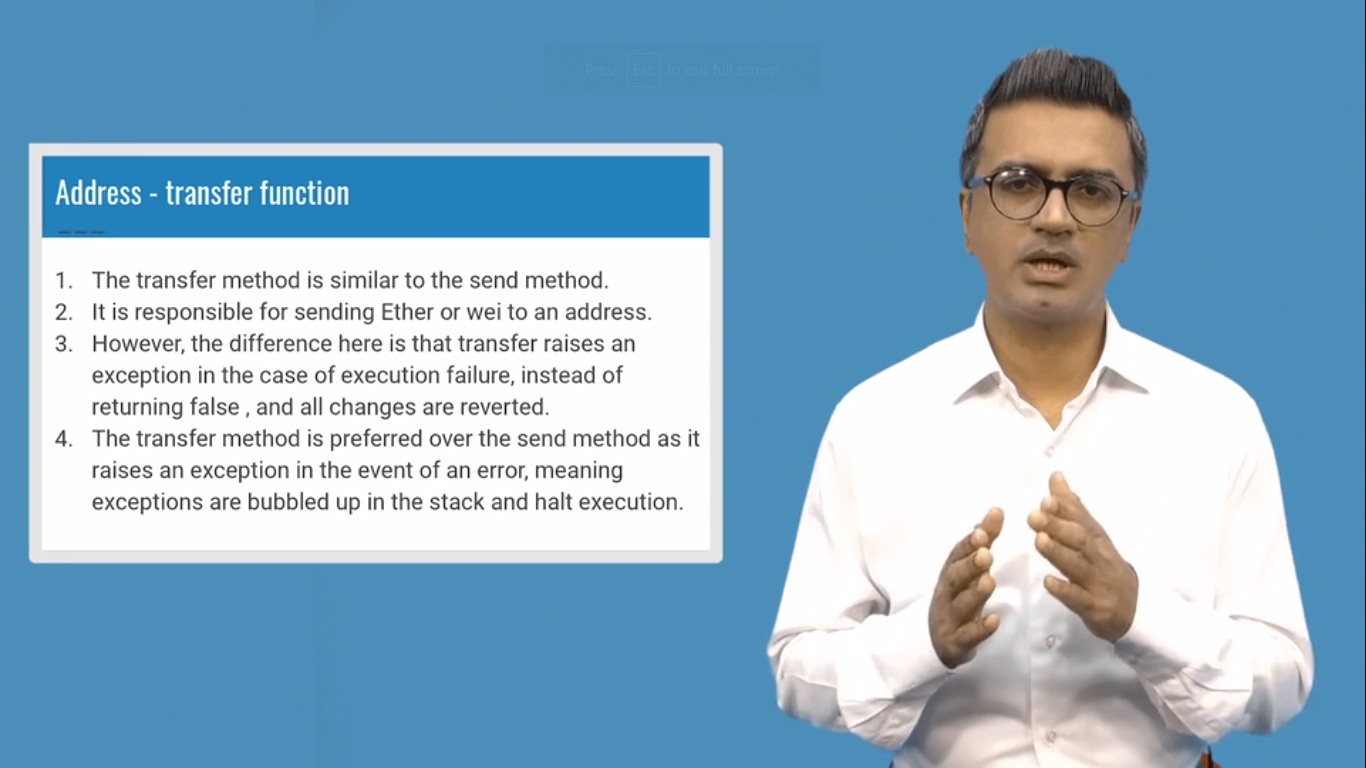
function testBalance() public payable returns(bool){

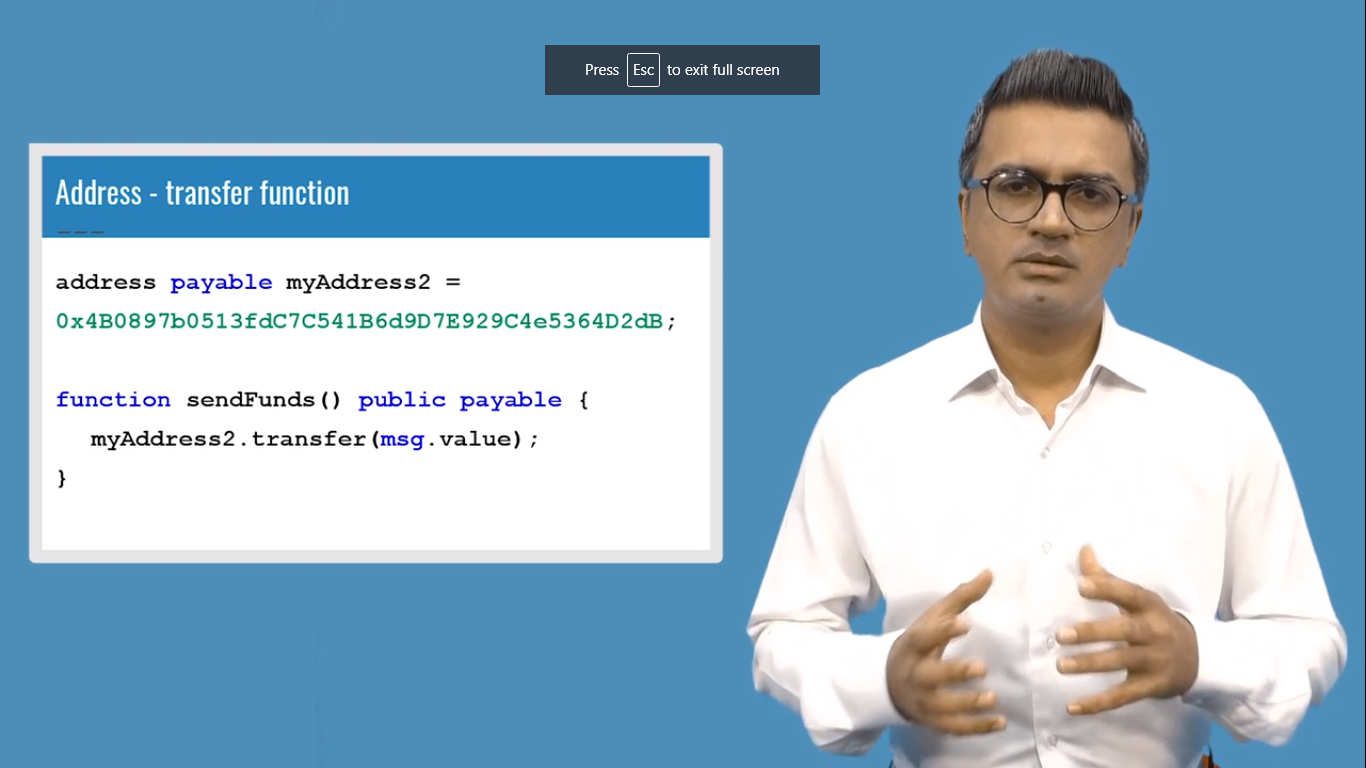
bool isSent = myAddress2.send(2 ether);

return isSent;

}

}





**Mapping:**

⇒ mapping is similar to hashing r dictionaries in other languages

⇒ used for key-value pair ( => )

⇒ mapping (int ⇒ string) m1;

⇒ **enum and struct can be used as value but not as key**

⇒ loop can not applied on mapping

⇒ map is only declared in state level (storage level) not in memory level

⇒ In function if you want to create mapping then you have to refer it with state level mapping

contract First{

mapping ( int => string ) m1;

function updateValue(int a,string memory val) public{

m1[a] = val;

}

function checkVal(int key) public view returns(string memory){

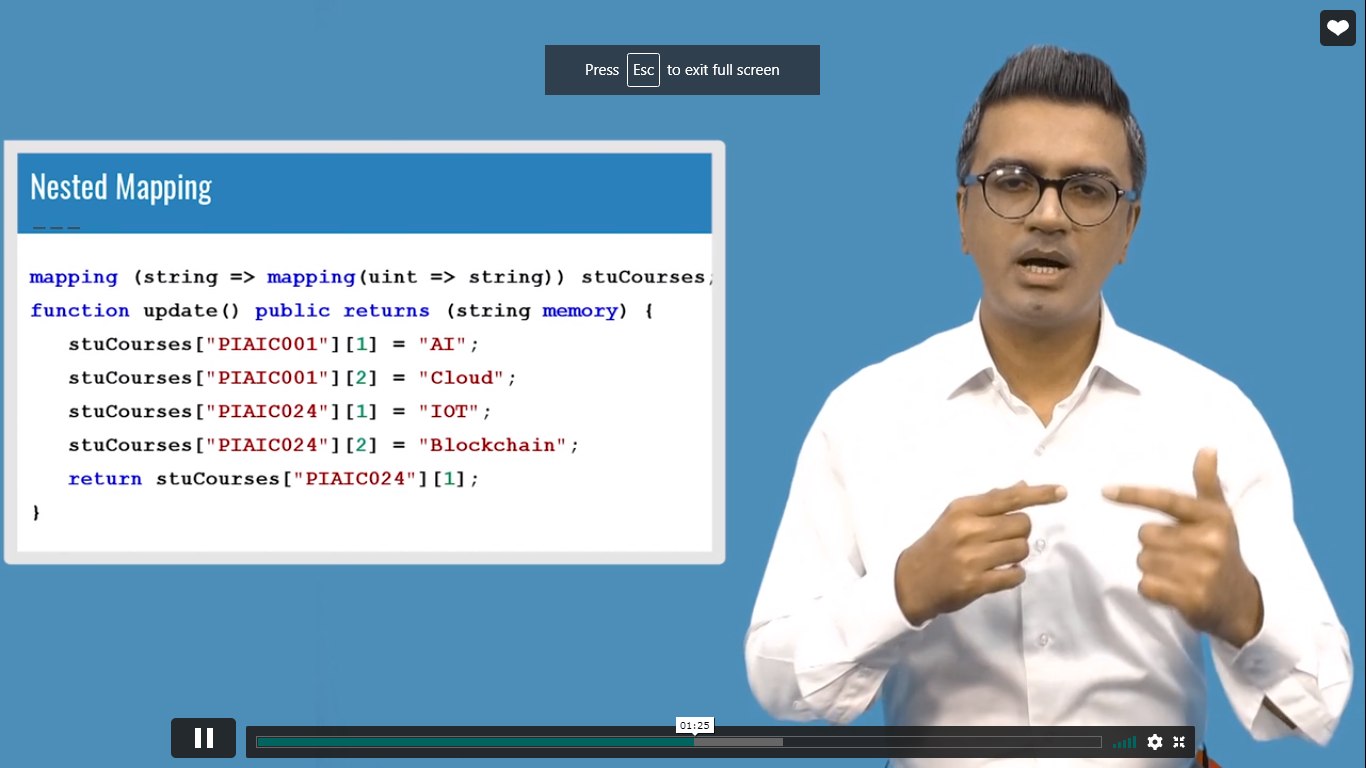
return m1[key];

}

}

⇒ mapping can not be receive in function return argument neither return from the function

**Nested Mapping:**

****