CST4125: Blockchain Development
Week: 16
Title: Solidity OOP

Dr Ian Mitchell

smerf.net
Bedfordshire,
UK

2023

Contact and Office Hours

Contact Details

- Name: Dr Ian Mitchell
- Room: TG10
- Address: Middlesex University, Computer Science, London, NW4 4BT
- email: smerf.net

Contact and Office Hours

Contact Details

- Name: Dr Ian Mitchell
- Room: TG10
- Address: Middlesex University, Computer Science, London, NW4 4BT
- email: smerf.net

Office Hours

- During term time only
- When: Autumn Term: Mondays 1100-1300hrs
- Please read notifications or emails
- There are occassions that these could be arranged online, e.g., due to industrial action or inclement weather

Deadlines

				Feedback	
Description	Submission	Weight	Deadline	Formative	Summative
1. Hyperledger	MyLearning	50%	18th December 2022	LW11-12	12/01/2023
2. Ethereum	MyLearning	50%	2nd April 2023	LW23-24	24/04/2023
Resits	MyLearning	50-100%	1st July 2023	None	None
Deferals	MyLearning	50-100%	1st July 2023	None	None

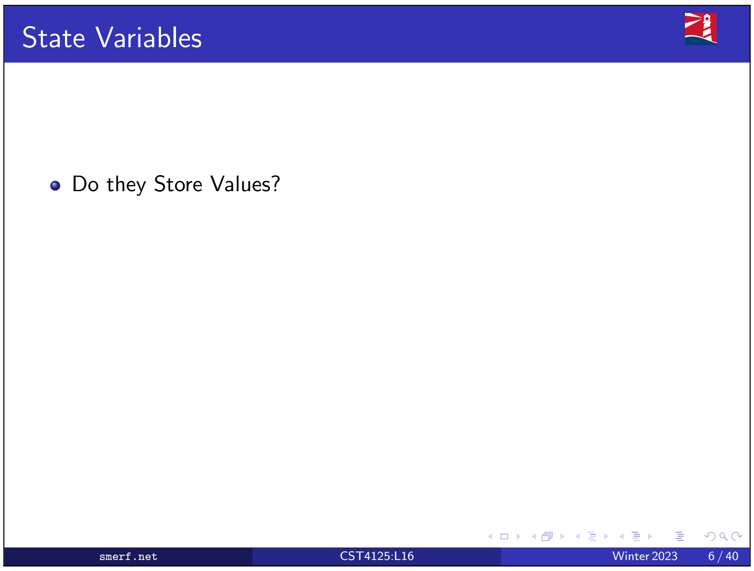
Lecture Aims

Aim

Much of the programming principles you will have come across in other languages. This emphasises the structure of solidity. This should be read with the previous lecture and together they give an overall view of the programming language solidity.
Much of this has been adapted from various textbooks [1, 2, 3].

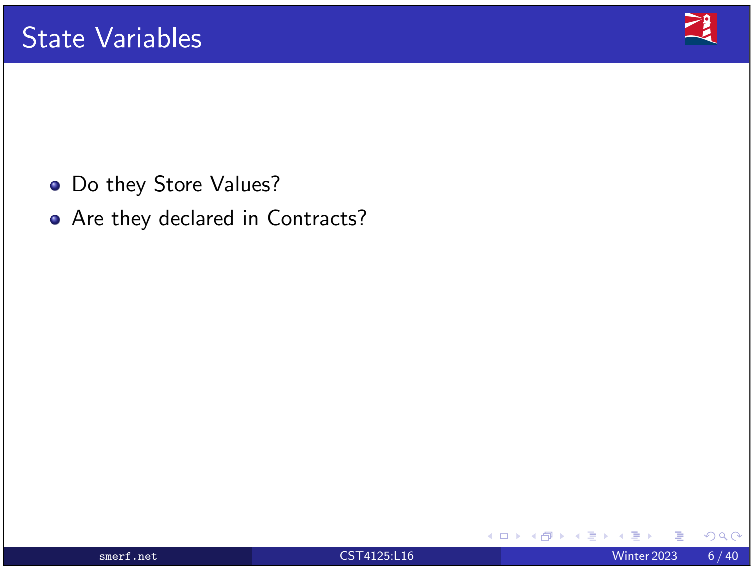
Lecture Objectives

- Structures
- Enumerators
- Arrays
- Mappings
- Iteration
- Control Structures
- Address
- Variable scope



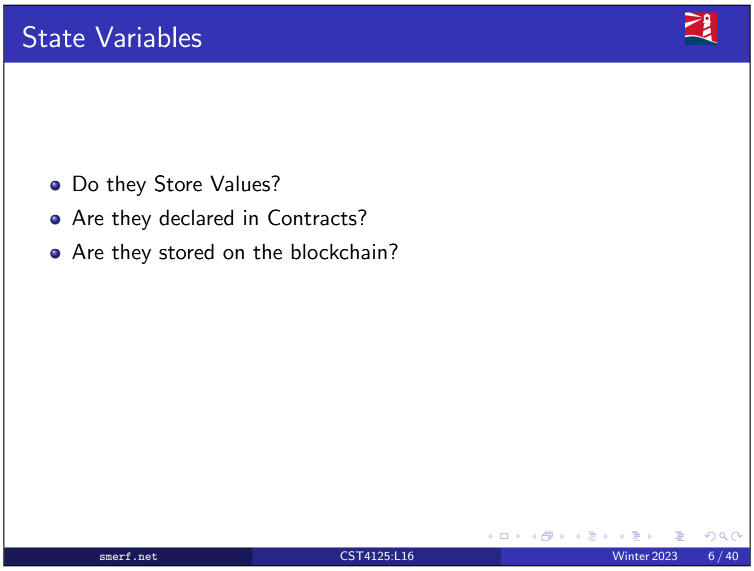
State Variables

- Do they Store Values?



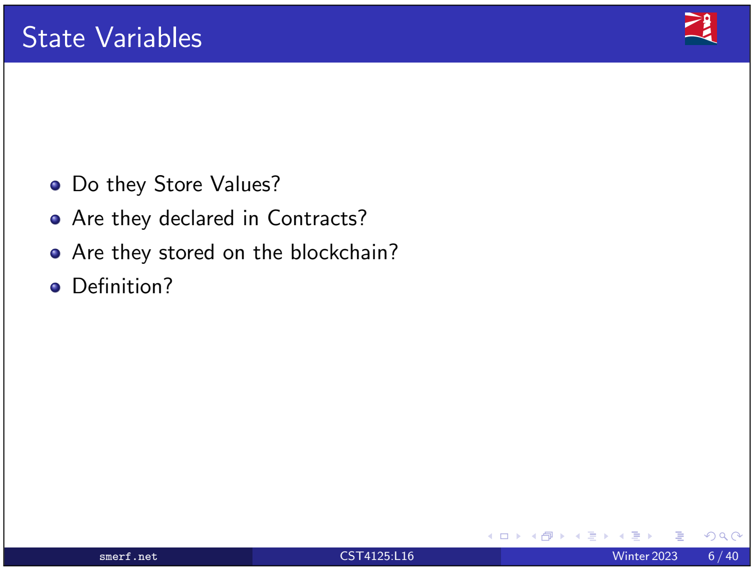
State Variables

- Do they Store Values?
- Are they declared in Contracts?



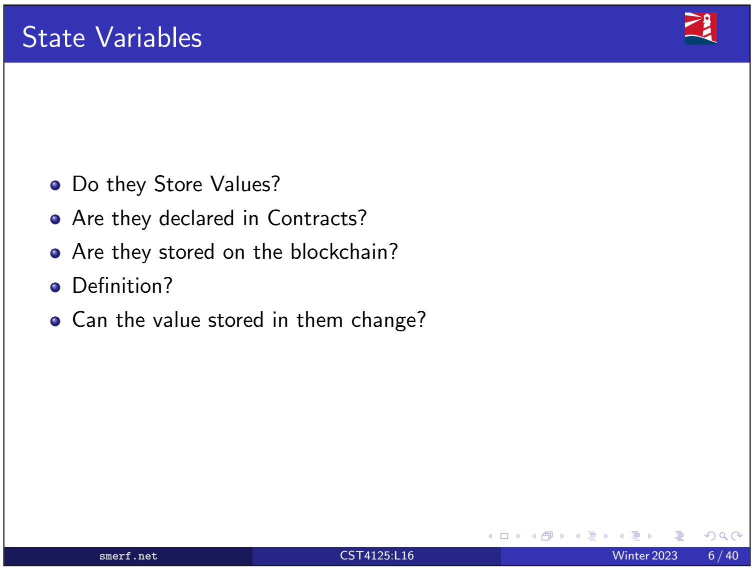
State Variables

- Do they Store Values?
- Are they declared in Contracts?
- Are they stored on the blockchain?



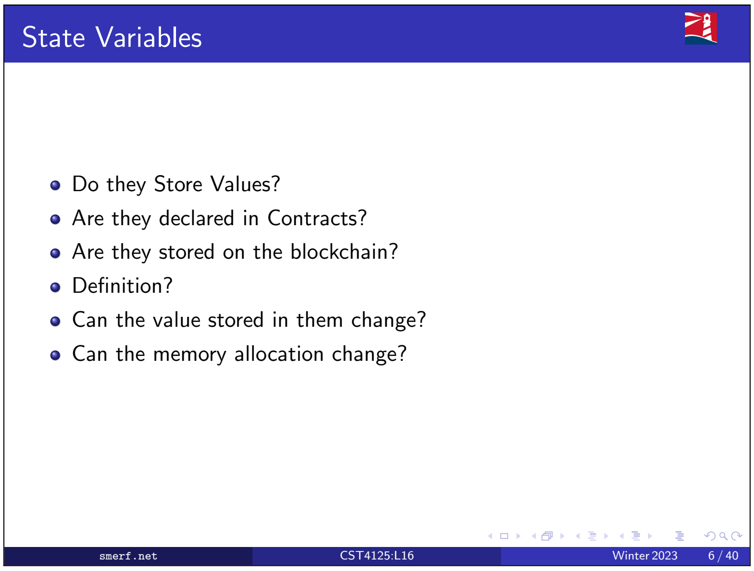
State Variables

- Do they Store Values?
- Are they declared in Contracts?
- Are they stored on the blockchain?
- Definition?



State Variables

- Do they Store Values?
- Are they declared in Contracts?
- Are they stored on the blockchain?
- Definition?
- Can the value stored in them change?



State Variables

- Do they Store Values?
- Are they declared in Contracts?
- Are they stored on the blockchain?
- Definition?
- Can the value stored in them change?
- Can the memory allocation change?

State Variables

- Do they Store Values?
- Are they declared in Contracts?
- Are they stored on the blockchain?
- Definition?
- Can the value stored in them change?
- Can the memory allocation change?
- Do they require a data type?

State Variable Qualifiers

internal

Default setting. Can be used within the current contract and functions. Can be used in inherited contract and functions. Cannot be accessed outside, however, can be viewed.

State Variable Qualifiers

internal

Default setting. Can be used within the current contract and functions. Can be used in inherited contract and functions. Cannot be accessed outside, however, can be viewed.

private

Similar to internal. Can only be used in contracts declaring them. Cannot be used in inherited contracts.

State Variable Qualifiers

internal

Default setting. Can be used within the current contract and functions. Can be used in inherited contract and functions. Cannot be accessed outside, however, can be viewed.

private

Similar to internal. Can only be used in contracts declaring them. Cannot be used in inherited contracts.

public

Makes them accessible. The compiler will create `getter` functions.

State Variable Qualifiers

internal

Default setting. Can be used within the current contract and functions. Can be used in inherited contract and functions. Cannot be accessed outside, however, can be viewed.

private

Similar to internal. Can only be used in contracts declaring them. Cannot be used in inherited contracts.

public

Makes them accessible. The compiler will create `getter` functions.

constant

Makes them immutable, variable must be assigned at declaration.

Enumerator Types

- Custom user-defined data type with a predetermined set of values
- Explicitly converted to `uint`
- First having zero, second one, and so on...
- `enum Gender {male, female, nonBinary}`
- What value is `Gender gender = Gender.female;`

Structure

```
enum Gender {female, male,
    nonBinary}

struct Person {
  string firstName;
  string lastName;
  uint2 age;
  Gender gender;
  //String address;
  }
```

Structure

- User-define data type
- Composite data type
- Consist of variables only, no code
- Instance of a structure
- `person = Person{"Ian", "Mitchell", "53", 0};`

Byte Data Type

```
contract example{
  bytes1 hex1 = 0xF0;
  bytes1 dec1 = 240;
  bytes1 neg  = -10;
  bytes1 char = 'a';
  bytes2 hex2 = 0x200;
}
```

Bytes

- Bytes refer to 8 bits signed integers
- Multiple bytes
- `bytes1, bytes2, ..., bytes32`
- fixed sized byte arrays
- value types
- default initial value: 0x00

Arrays revisited

```
contract example{
  int[4] ia1;
  int[]  ia2;
  byte[4] ba1;
  int[2] = [int(1), 2];
  int[] = new int[](2);
  int[] = [int(1), 2];
}
```

Syntax

- Store hex, why?
- Unsigned Integer, max. value for 1 byte?
- Signed Integer
- multiple bytes
- Store outside the range, compile error.

Array properties

index : access individual elements. Not supported on Strings. Write to arrays, except the string type and fixed sized byte arrays.

push : is only supported by dynamic arrays.

length : viewing/reading is supported by all arrays, except string types. Modifying is supported by dynamic arrays and bytes

Byte arrays

```
contract example{
  bytes byteArray1;
  byte[] byteArray2;
  byteArray1 = "Ian
    Mitchell";
  byteArray1 = new bytes
    (100);
  :
  :
  byteArray1.push(byte
    (200));
  return  byteArray1.
    length();
  bytesArray1.length = 8;
}
```

`bytes v. byte[]`

- `bytes`
- `byte[]`
- `bytes` access length property
- `bytes` access pop & push methods
- `byte[]` can also access these

Real Numbers

- `ufixed, fixed`
- not supported in current version

String Array

```
contract example{
  String name = "Ian Mitchell";
  Bytes byteName = bytes(name);
}
```

Strings

- Strings are dynamic data types
- Based on byte arrays
- Strings cannot be:
  - pushed
  - popped
  - access length property
- Overcome this?

Mappings

```
contract example{
  mapping ( uint => address ) scores;
}
```

Mappings

- Ubiquitous throughout solidity contracts
- Similar to hash tables or dictionaries
- Store key-value pairs
- Retrieve values based on key, not index
- `mapping ( keyType => valueType ) x;`
- No iterator
- Only declared as a state variable

Mapping

Behaving like an Array

```
//SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract mapping1{
    mapping (uint => address) addressBook;
    uint index;

    function addAddressBookEntry( address entry) public returns (uint){
        index+=1;
        addressBook[index]=entry;
        return index;
    }

    function getAddressBookEntry(uint idx) public view returns (address){
        return addressBook[idx];
    }
```

Convert a Mapping

```
    function convertToArray() public view returns (address[] memory){
        address[] memory addressArray = new address[](index);
        for (uint i=1; i<index; i++){
            addressArray[i-1] = addressBook[i];
        }
        return addressArray;
    }
```

Arrays v. Mappings

Arrays

- Static and Dynamic
- Avoid Dynamic
- Iterators
- `push, pop, length`
- Have both
- Retrieval

Mappings

- reference type
- Always use a mapping, unless you cannot
- No iterators
- key - value pairs
- Retrieval
- Size
- Conversions are expensive and cost more gas, be careful.
- Nested mapping

Date

Unit	Calculation	Seconds
Day	24*60*60	86,400
Hour	60 * 60	3,600

Table: Seconds in a Day and Hour

- Unix Timestamp
- Number of seconds since 1st Jan 1970
- 32-bit unsigned integer
- Limitations: 19th Jan 2038
- time.is

Variables

Hoisting

- Default values
- Static type
- Declare variables at end of function?
- un-Hoist in later compilers

Scope

- Contract-level global variables
- Function-level local variables
- State variables:
  - public
  - internal
  - private

Variable conversion

Type conversion

- `uint8 x=512;`
- `byte y=512;`
- Implicit conversion
- No need for an operator
- No loss of data
- Convert small to large
- `uint8 x=0xFF;`
- `uint16 y=0xFFFF;`
- `x=y;`
- `y=x;`
- `uint32 z;`
- `z=x+y;`

Explicit type conversions

- Due to loss of data
- Range
- `uint` ≠ `-1;`
- `uint8` ≠ `0x100;`
- `uint8` ≠ `uint16;`
- Avoid address conversion

Explicit Type Conversions

```
int  y = -2;
uint x = uint(y);//0
   xffffffffffffffffffffffffffffffffffffffffffffffffffff

     //64 Hex chars (2*32)
     //-3 in two's complement
     // flip and add 1
uint32 a = 0x12345678; // 4 bytes, 32 bit
uint16 b = a;    // b will be 0x5678
    // high order bits truncated
uint16 d = 0xFFFF;
uint32 e = d;   //d = 0x0000FFFF
    // padded 0s for high order bits
assert (d==e);


```

Explicit byte Conversion

```
bytes2 a = 0x1234;  //16-bit number
bytes1 b = bytes1(a);   // b = 0x12
       // truncate sequence
bytes2 d = 0xfedc;
bytes4 e = bytes4(a); // e = 0xfedc0000
       // padded 0s on right
assert(a[1] == b[0]);
assert(d[0] == e[3]);
assert(d[1] == e[2]);
```

Explicit byte to uint Conversion

```
bytes2 a = 0xfedc;
uint32 b = uint16(a);   // bytes2 == uint16
        // b = 0x0000fedc;
uint64 c = a;   // fail
uint64 c = uint64(bytes4(a));
      // convert 2 to 4 byte
      // convert 4 byte to 64-bit
uint8 d = uint8(uint16(a));
      // d = 0xdc;
uint8 e = uint8(bytes1(a));
      // e = 0xfe;
```

Literals conversion

```
bytes2 a = 100;   //error, must populate all bytes
bytes2 b = 0x123;   //error, ditto
bytes2 c = 0xFF;  //error, ditto
bytes2 d = 0xFFFF;  // compile
bytes2 e = 0x00FF;  // compile
bytes2 f = 0;   // exception, allows zero
bytes2 g = 0x0;   // exception, ditto
```

Comparison Operators

Operators	Description	Example
==	Equals	x==100;
!=	Not equals	x!=100;
>	Greater than	x>100;
<	Less than	x<100;
>=	Greater than and equal to	x>=100;
<=	Less than and equal to	x<=100;

Table: Comparison Operators in Solidity

Logical Operators

Operators	Description	Example
&&	AND	(x>10) && (x<20)
||	OR	(x<10) || (x>20)
!	NOT	!(x==10)

Table: Logical Operators in Solidity

Bitwise Operators

Operators	Description	Example
∼	bitwise NOT	∼x
>>	shift right	x>>2
<<	shift left	x<<2
&	bitwise AND	x&y
|	bitwise OR	x|y
∧	bitwise XOR	x∧y

Table: Bitwise Operators in Solidity

Mathematical Operators

Operators	Description	Example	Order
++	Postfix increment	i++	1
--	Postfix decrement	i--	1
++	Prefix increment	++i	2
--	Prefix decrement	--i	2
**	Exponentiation	x**3	3
*	Multiplication	x*4	4
/	Division	x/4	4
%	Modulo	x%4	4
+	Addition	x+5	5
-	Subtraction	x-5	5

Table: Mathematical Operators in Solidity, showing precedence in right column.

Assignment Operators

Operators	Description	Example
<<=, >>=	shift assignment	x<<=2
+=, -=	add, subtract assignment	x+=2
*=, /=, %=	multiply, divide, modulo assignment	x*=2
=	assignment	x=2
| =, ∧ =, &=	Bitwise assigment	x&=2

Table: Assignment Operators in Solidity, order of precedence is 15

While Loop

```
while ( expression is true) {
  statement;
  statement;
}
```

Do ... While Loop

```
initialise counter value;
do {
  statement;
  change counter value;
} while ( condition is true);

uint i=0;
do {
  i++;
} while( i<10 );

```

For Loop

```
for( initialise counter value; stopping condition; change counter value){
  statement;
  statement;
}
```

If control

```
if( condition/expression is true) {
  statement;
  statement;
}
else if ( this condition/expression is true) {
  statement;
  statement;
}
else {
  statement;
  statement;
}
```

Break statement

```
//SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract A{
  uint max=3;

  event display(uint);

  function loopy() public  {
    for(uint i=0; i<10; i++){
      if( i==max )
        break;
      emit display(i);
    }
  }
}
```

Continue statement

```
//SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract A{
  uint max=3;

  event display(uint);

  function loopy() public  {
    for(uint i=0; i<10; i++){
      if( i<=max )
        {continue;}
      emit display(i);
    }
  }
}
```

Summary

- Variables
- Data Types
- Operators
- Programming
- Complete these exercises and ready to start writing simple smart contracts

Reading

- Read Chapter 4 in [4]

References I

[1] Andreas M. Antonopoulos and Gavin Wood. *Mastering Ethereum*. 1st. O'Reilly, 2018.

[2] R. Infante. *Building Ethereum Dapps*. Manning, 2019. ISBN: 9781617295157.

[3] Ritesh Modi. *Solidity Programming Essentials*. Packt, 2018. ISBN: 978-1-78883-138-3.

[4] X. Wu, Z. Zhihong, and D. Song. *Learn Ethereum*. 1st. Packt, 2019. ISBN: 9781789954111.