Lecture 2

Solidity Fundamentals

Disclaimer - many fish in the sea!

Solidity File Structure

Inside a .sol source file

- SPDX-License-Identifier
  - https://spdx.dev/
  - Can have an "unlicensed" identifier
- pragma
  - solidity version (to match compiler)
  - ABI encoder / decoder
  - Experimental pragmas: ABIv2, SMTchecker (Formal Verification)
- Import
- Comments
  - Single line: //
  - Multi line: /* */
  - Natspec

npm version semantic

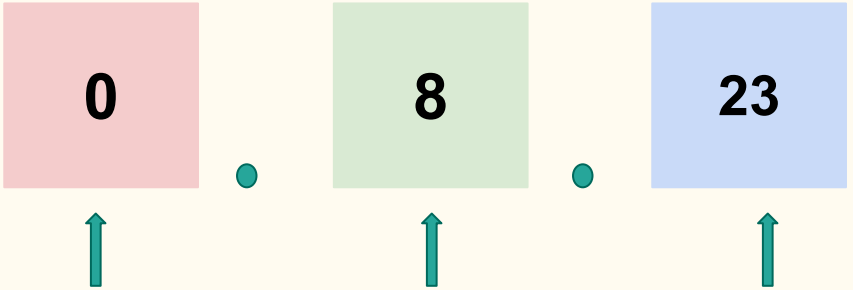
Solidity has not even hit 1 stable release yet!!!

^

use this specific version

>= <

range of versions to use



Major Release

Breaking changes

Minor Release

Features Backwards Compatible

Patch

Bug Fix Backwards Compatible

Importing

Virtual Filesystem on the Compiler

Initial files (plus dependencies) can be loaded on CLI or JSON format.

Compiler can add other files during compile time

Direct Import

```
import "/project/lib/util.sol";

import "lib/util.sol";

import
"@openzeppelin/address.sol";

import
"https://example.com/token.sol";
```

Relative Import

```
import "./";

import "../";
```

Natspec - Natural Language Specification Format

Tag		Context
`@title`	A title that should describe the contract/interface	contract, library, interface
`@author`	The name of the author	contract, library, interface
`@notice`	Explain to an end user what this does	contract, library, interface, function, public state variable, event
`@dev`	Explain to a developer any extra details	contract, library, interface, function, state variable, event
`@param`	Documents a parameter just like in Doxygen (must be followed by parameter name)	function, event
`@return`	Documents the return variables of a contract’s function	function, public state variable
`@inheritdoc`	Copies all missing tags from the base function (must be followed by the contract name)	function, public state variable
`@custom:...`	Custom tag, semantics is application-defined	everywhere

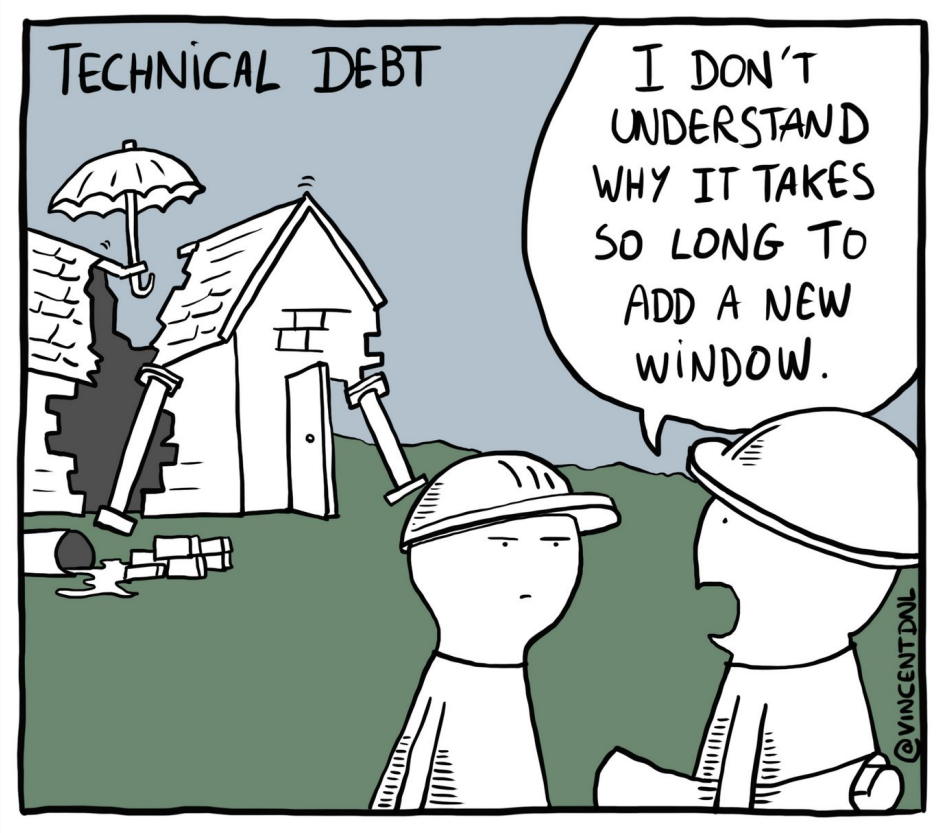
Natspec - An Example

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.2 < 0.9.0;

/// @title A simulator for trees
/// @author Larry A. Gardner
/// @notice You can use this contract for only the most basic simulation
/// @dev All function calls are currently implemented without side effects
/// @custom:experimental This is an experimental contract.
contract Tree {
    /// @notice Calculate tree age in years, rounded up, for live trees
    /// @dev The Alexandr N. Tetearing algorithm could increase precision
    /// @param rings The number of rings from dendrochronological sample
    /// @return Age in years, rounded up for partial years
    function age(uint256 rings) external virtual pure returns (uint256) {
        return rings + 1;
    }

    /// @notice Returns the amount of leaves the tree has.
    /// @dev Returns only a fixed number.
    function leaves() external virtual pure returns(uint256) {
        return 2;
    }
}
```

Source:

https://docs.soliditylang.org/en/v0.8.17/natspec-format.html

Technical debt - the cost of bad code



Solidity Conventions

Max Line Length = 120 char
Breakdown new lines uses tabs
	`thisFunctionCallIsReallyLong(`
	`  longArgument1,`
	`  longArgument2,`
	`  longArgument3`
	`);`

Encoding	UTF-8 or ASCII

Import Statements
Always at top after license identifier and pragma
	`// SPDX-License-Identifier: MIT`
	`pragma solidity >=0.4.0 <0.9.0;`
	
	`import "./Owned.sol";`

Whitespace
No space between brackets/quotes
Space around operators
	`spam(ham[1], Coin({name: "ham"}));`
	
	`x = 1;`
	`y = 2;`

Solidity Conventions - Naming

Contracts, Libraries, Interfaces, Structs, Events	`CapWords`
	
	`contract MyContract{}`
	`struct PersonStruct{}`

Function Names, Function Arguments, Variable Names	`mixedCase`
	
	`int myInteger;`
	`function helloWorld();`

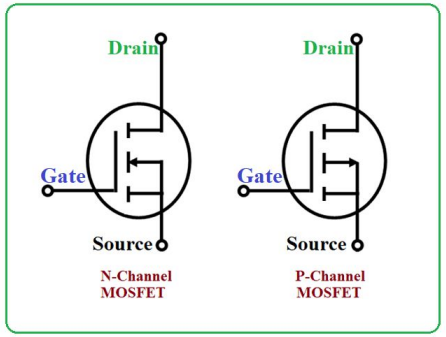
Constants	`ALLCAPS`
	
	`int WINNING_NUMBER = 5;`

Primitives

Value Types

Primitives - integers

At the most basic level, computers operate on 1 and 0 - This system is called **Binary**.



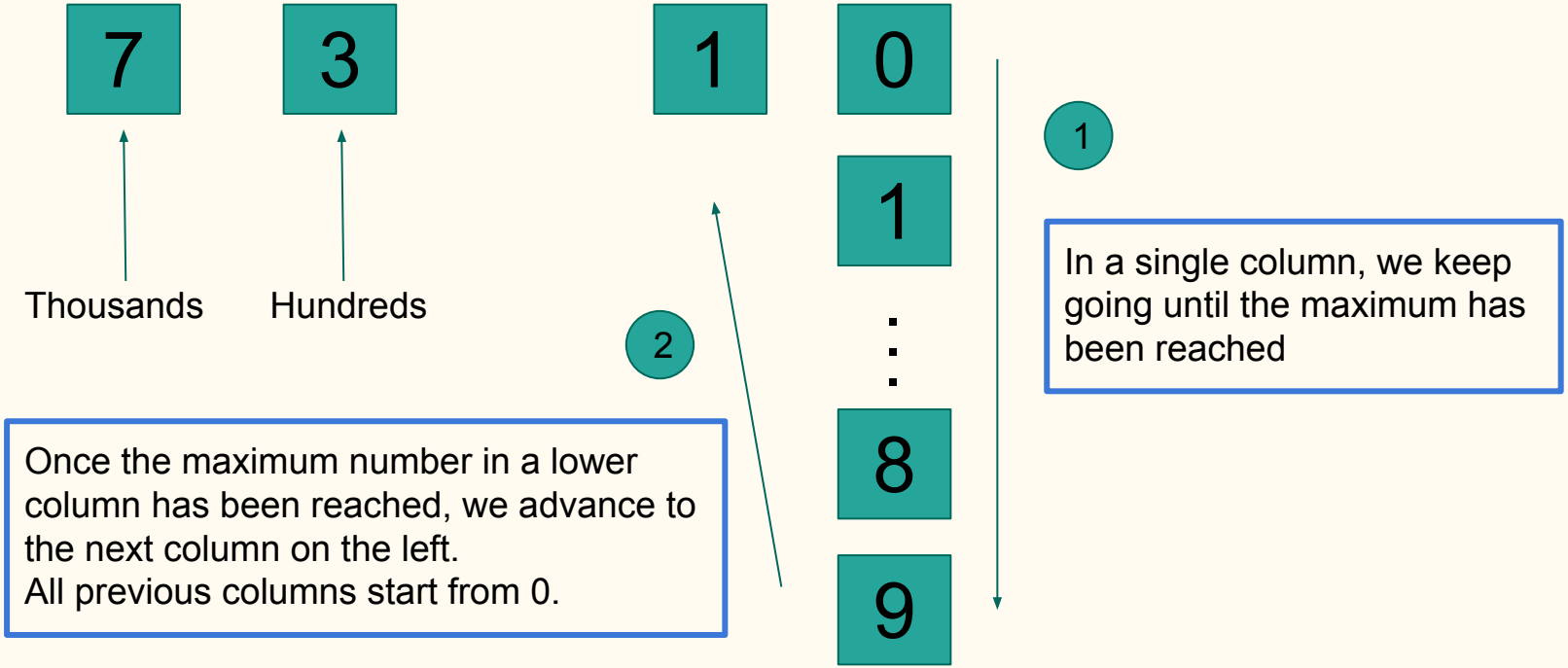
From a hardware perspective:
High voltage (5V) = "1"
Low voltage (0V) = "0".
Currently, these are the only 2 possible states and why computers are binary in nature.

Quantum computing aims to break this constraint!

Decimal	Hexadecimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

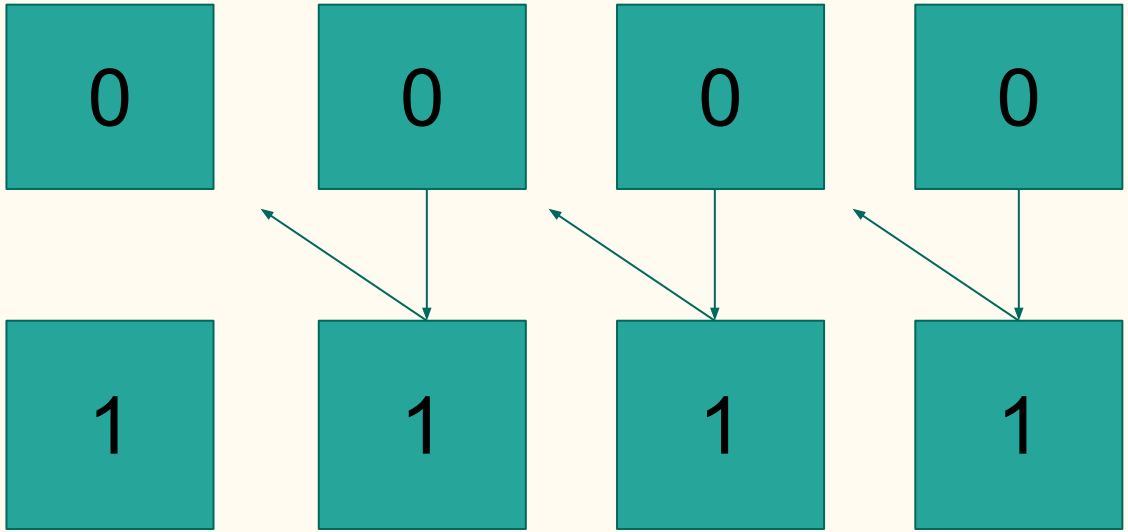
Primitives - integers

Let's take a look at the Decimal System



Primitives - integers

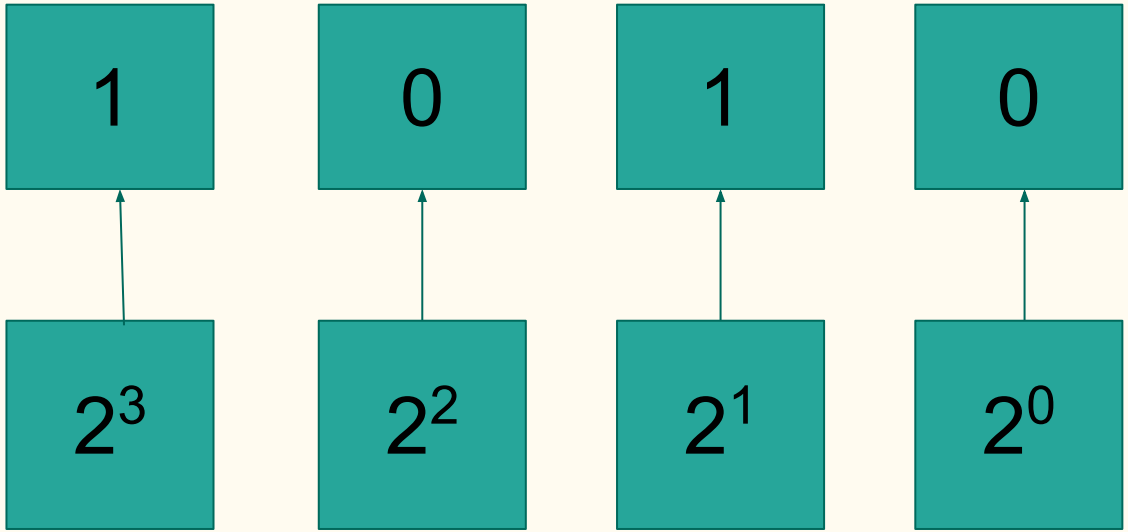
Same intuition for a Binary System



Decimal	Hexadecimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Primitives - integers

Binary to Decimal -> true or false

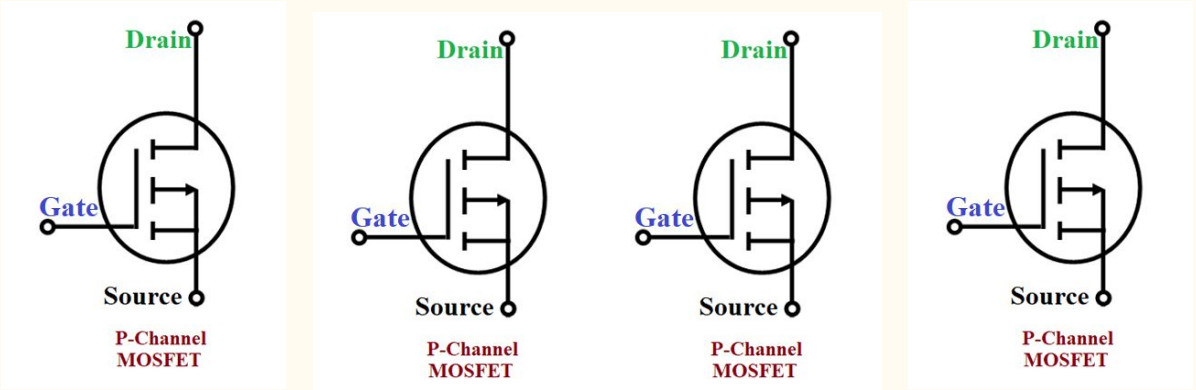


What is this in Decimal?

$2^3 + 2^2 + 2^1 + 2^0$
$= 2^3 + 2^1$
$= 8 + 2$
$= 10$

Primitives - integers

bits and Bytes



1	0	1	0

Each storage unit is a bit.

There are 4 bits here

8 bits = 1 Byte

Note: KB -> MB -> GB -> TB -> PB is not $10^3 = 1000$ but $2^{10} = 1024$ intervals!

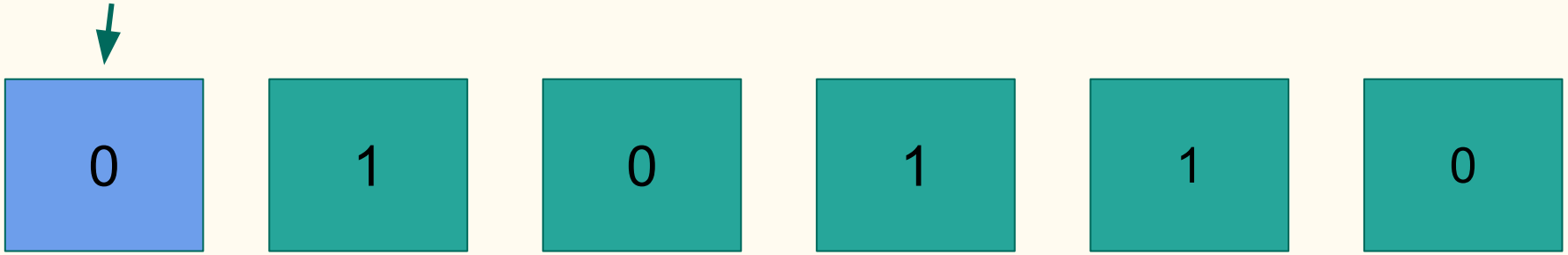
Primitives - integers

uint8	uint16	uint32	uint64	uint128	uint256
int8	int16	int32	int64	int128	int256

No Decimals!

Primitives - integers

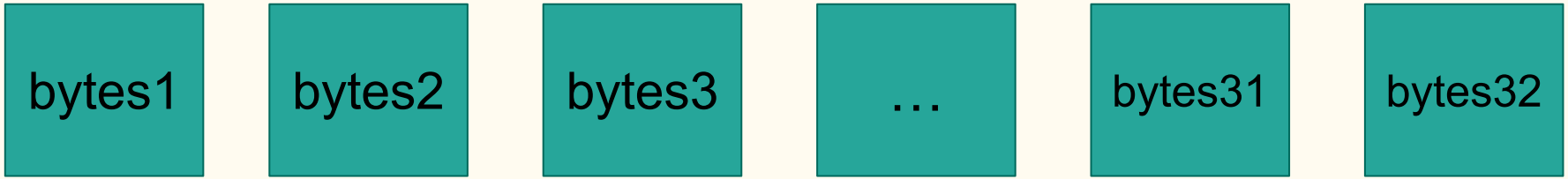
The first bit becomes the "sign" -> 0 is a positive number, 1 is negative.



Since 1 bit is taken up to mean the sign, remember you can only have numbers half as big as unsigned integers

Primitives - Bytes

8 **b**its = 1 **B**yte



UTF8 Encoding

This is unicode. The OG encoding.

As encoding formats expanded to include more scripts and even emojis, there was:
UTF8 → UTF16 → UTF32...
ASCII and more

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

Primitives - Boolean

Primitives - Addresses

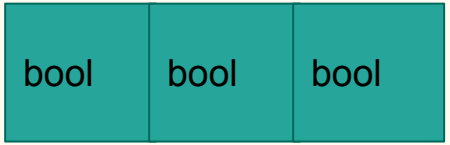
0xFd348ab656a6127f4280C5b1218D46D80a41e224

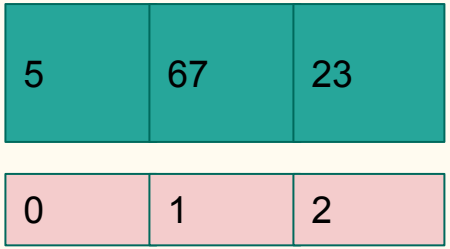
20 Bytes = 160 bits

Arrays, Mapping, String, Struct

Reference Types

Type - Array



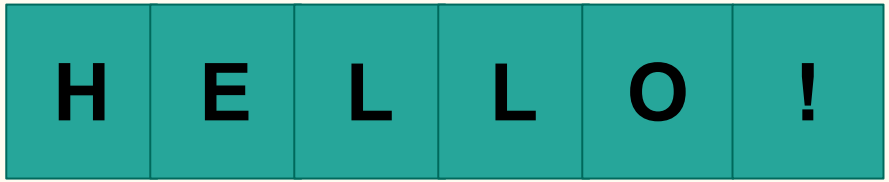


array[0] = 5
array[1] = 67
array[2] = 23

indexes start at 0!

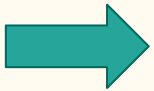
array.push	**array.length**
array.pop	**delete array**

Type - String



Type Casting

Solidity Strings have no functions!!!



```
string hello = "hello";
bytes casted_hello = bytes(hello);
```

```
uint8 a = 1;      =>    00000001
b = uint16(a);    =>    0000000000000001
```

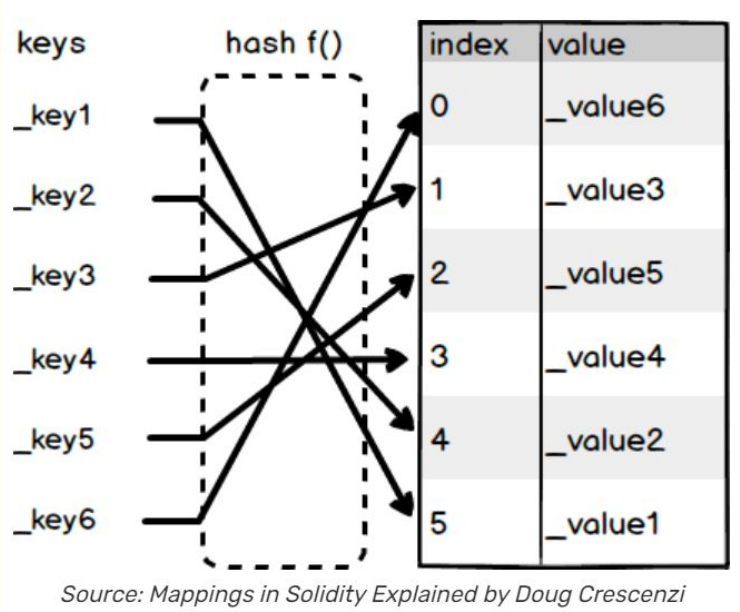
What happens when we go from uint16 to uint8?

Type - Mapping

Key => **value**

Now it's getting annoying…

- can't find length
- can't loop through keys

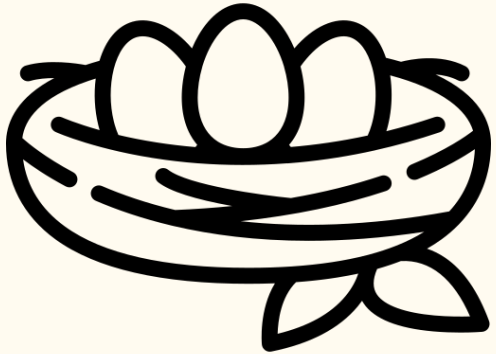


Source: Mappings in Solidity Explained by Doug Crescenzi

Type - Struct

```
struct Object {

    property1;
    property2;

}
```

```
Object.property
```



Nesting allowed!

Observe tight variable packing

https://fravoll.github.io/solidity-patterns/tight_variable_packing.html

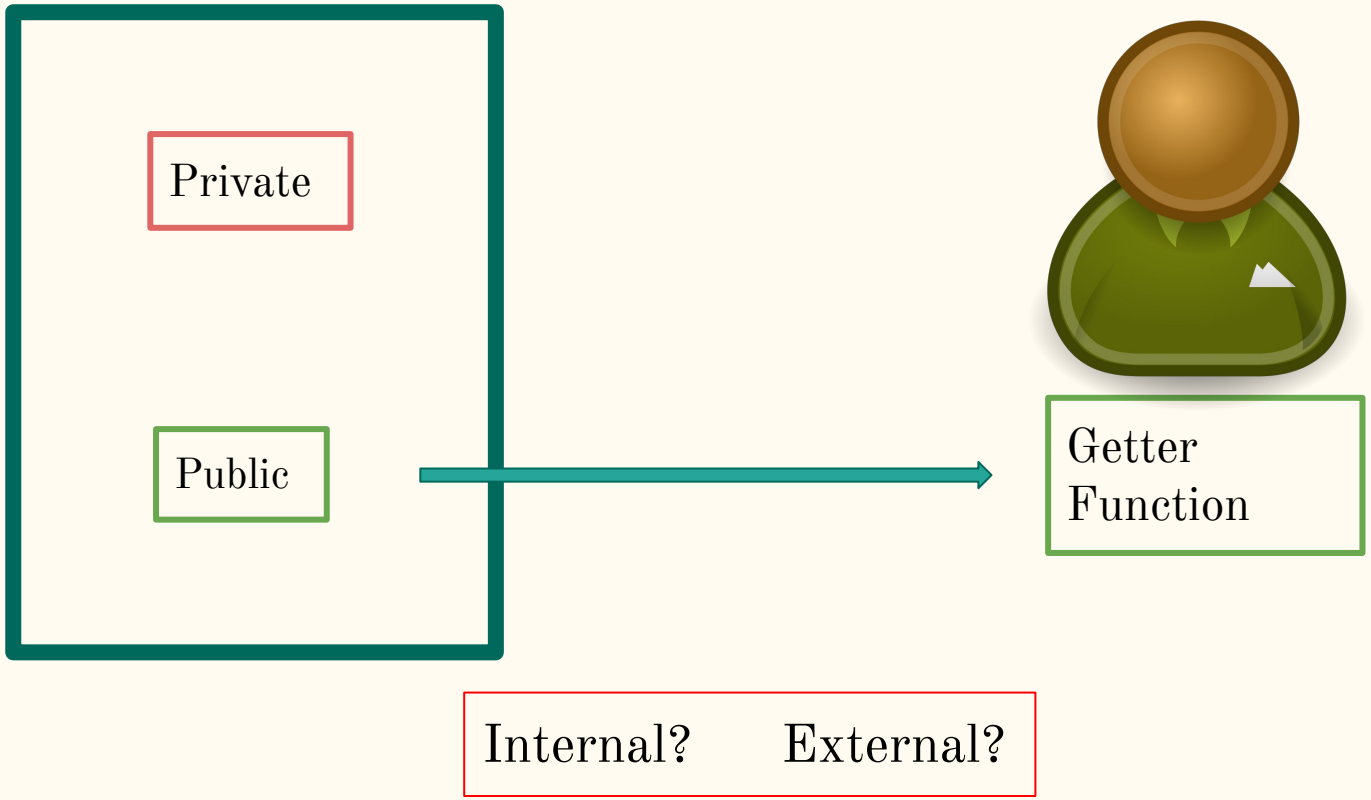
Solidity Variables

Instantiation and Scope

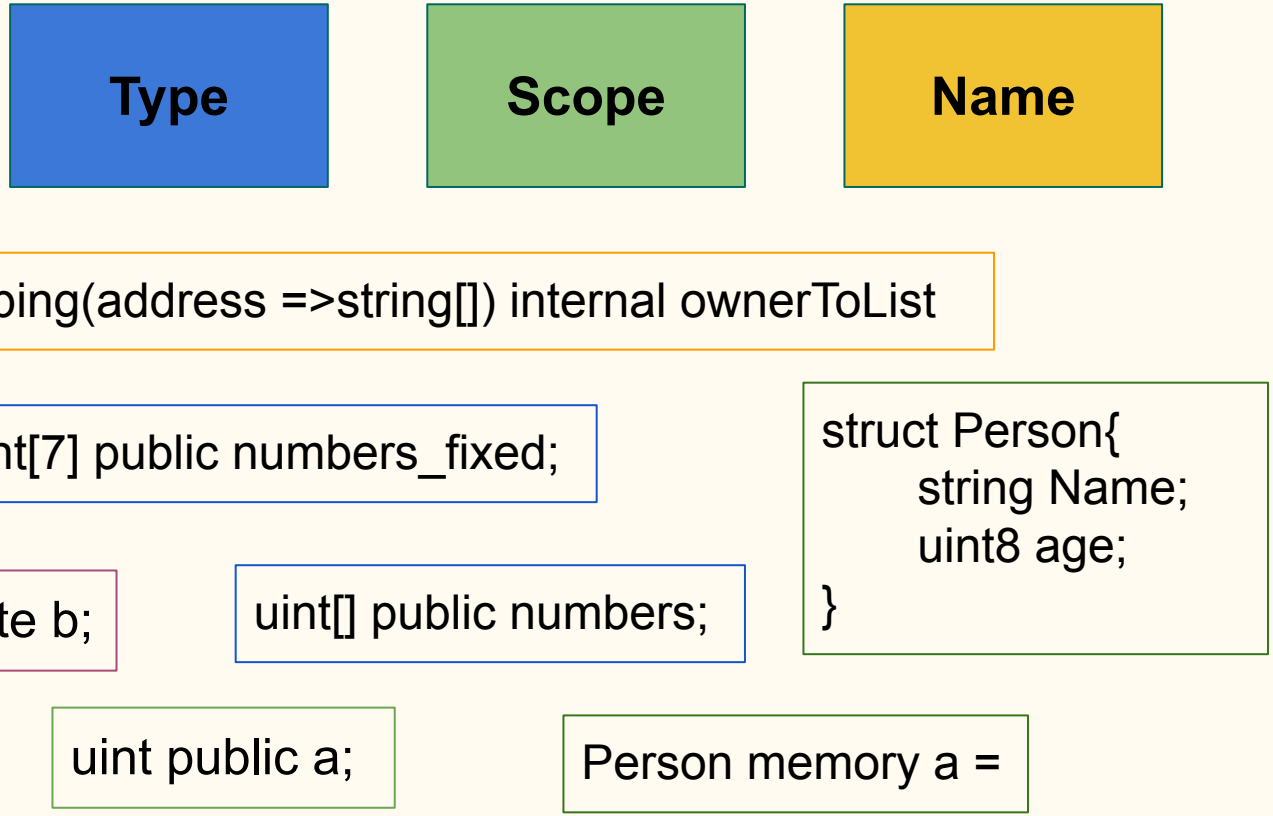
Existence is..... dynamic and fixed / variable and literal

Dynamic	Can only do if in storage, expensive and painful	int[] fixed;
Fixed	Amount of memory needed known upon declaration	int[5] fixed; int[] fixed = new int(5);
variable	Only the type is known	int a;
literal	type and value known	int a = 5;

Scope



Instantiation



Operators

Algorithmic, Relational, Logical

Operators

Algorithmic	Relational	Logical
+ - % *	==	&&
++ - -	< > <= >=	||
%	!=	!

Flow Control

if, for, while

If ....else

```
if (condition){
    execution when condition is true
} else {
    execution for all cases when condition is false
}
```

for loop

```
for (initialize counter; condition of counter; increment counter) {
    continue executing until condition is met;
}
```

```
for (uint i = 0; i < 10; i++){
    start i from 0, do thing until i is 9 and i increases by 1 each loop;
}
```

while loop

```
while (condition) {
    continue execution until condition becomes false
}
```

Break - get out of loop now!

Continue - skip the reminder of the execution, go to next iteration

Decimal to Binary Converter

putting it all into practice

Process Flow

Convert 13_{10} to binary:

Division by 2	Quotient	Remainder	Bit #
13/2	6	1	0
6/2	3	0	1
3/2	1	1	2
1/2	0	1	3

So $13_{10} = 1101_2$

1. Loop through decimal number
2. Get its Quotient & Reminder
3. Store Remainder
4. Flip Remainder array and turn into string
5. Return result