Lecture 3

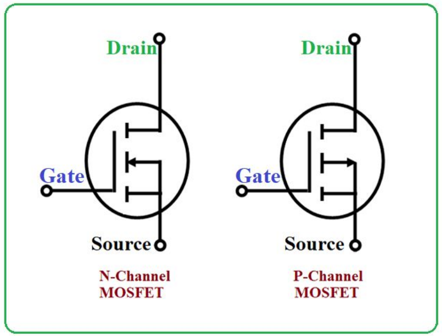
Solidity Basics

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

7 3 1 0

Thousands Hundreds

1

In a single column, we keep going until the maximum has been reached

1

:

8

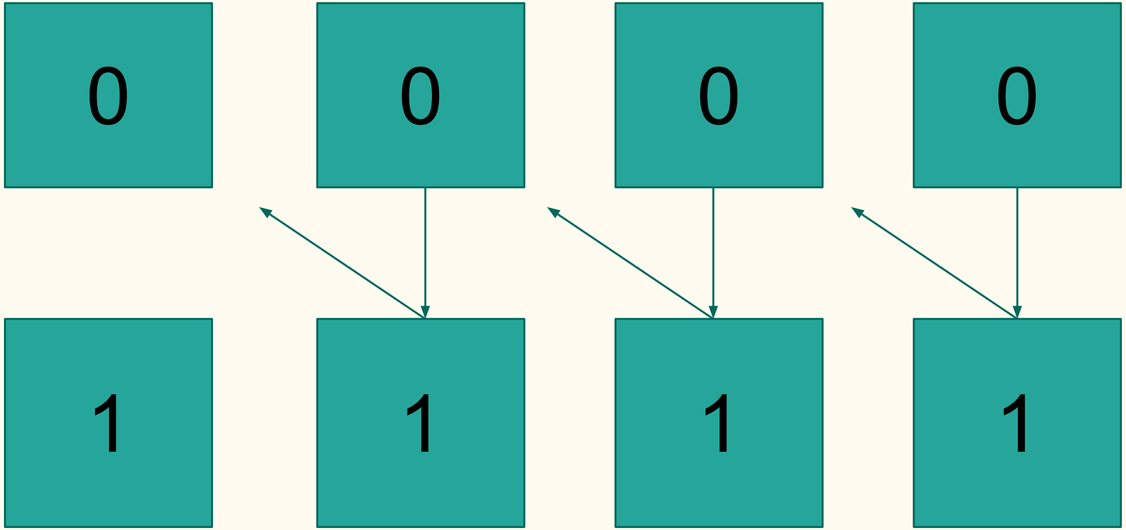
9

2

Once the maximum number in a lower column has been reached, we advance to the next column on the left.
All previous columns start from 0.

Primitives - integers

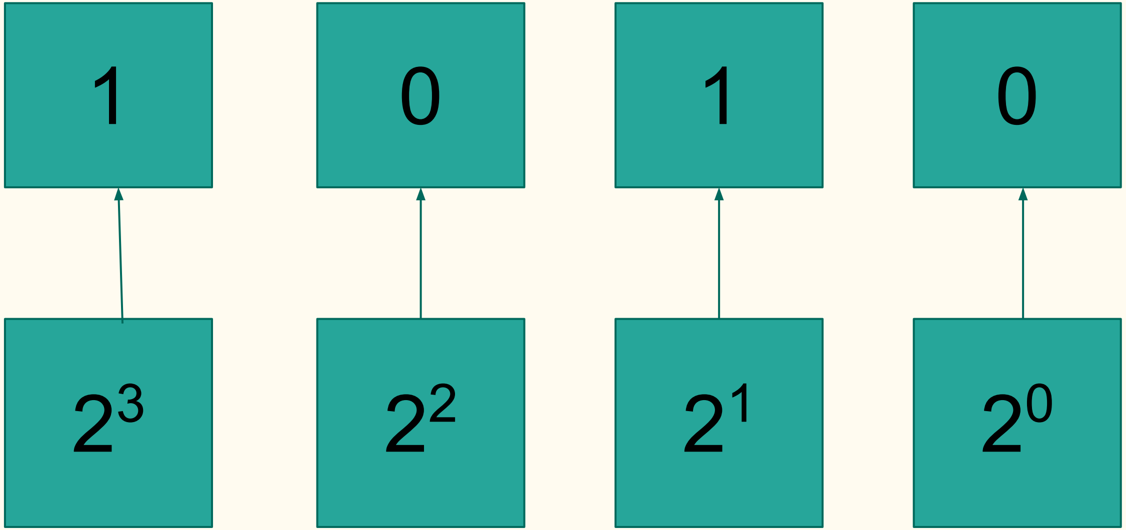
Same intuition for a Binary System



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

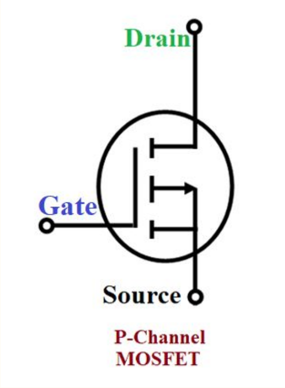


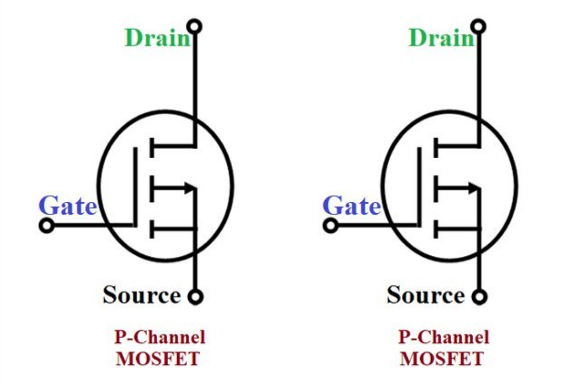
What is this in Decimal?

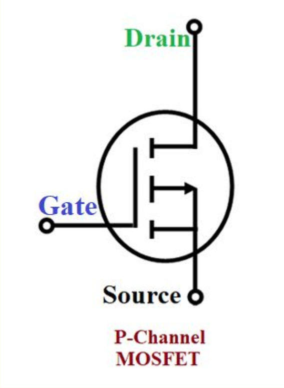
$$
\begin{aligned}
&2^3 + \cancel{2^2} + 2^1 + \cancel{2^0} \\
&= 2^3 + 2^1 \\
&= 8 + 2 \\
&= 10
\end{aligned}
$$

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^{10} = 1000 but 2^{10} = 1024 intervals!

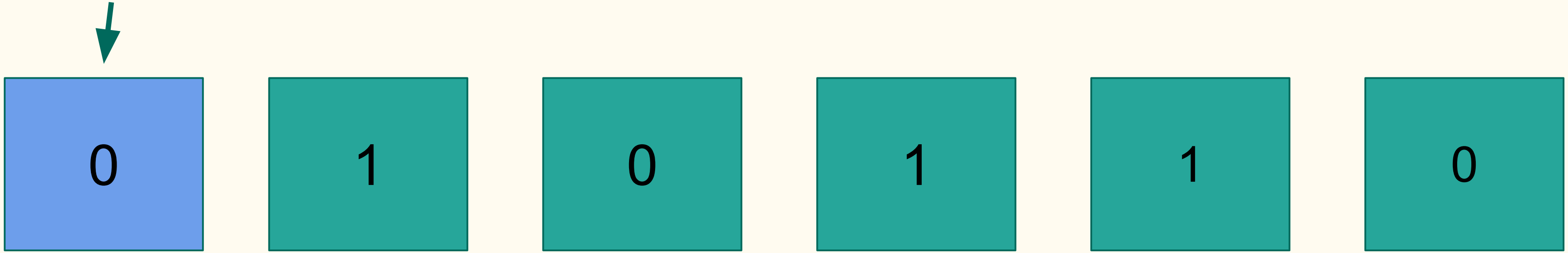
Primitives - integers

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

bytes1 | bytes2 | bytes3 | … | bytes31 | bytes32

Primitives - Boolean

Primitives - Addresses

0xFd348ab656a6127f4280C5b1218D46D80a41e224

20 Bytes = 160 bits

Arrays, Mapping, String, Struct

Reference Types

Type - Array

int	int	int

bool	bool	bool

5	67	23
0	1	2

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

indexes start at 0!

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

Type - String

H	E	L	L	O	!

Type Casting

Solidity Strings have no functions!!!

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

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

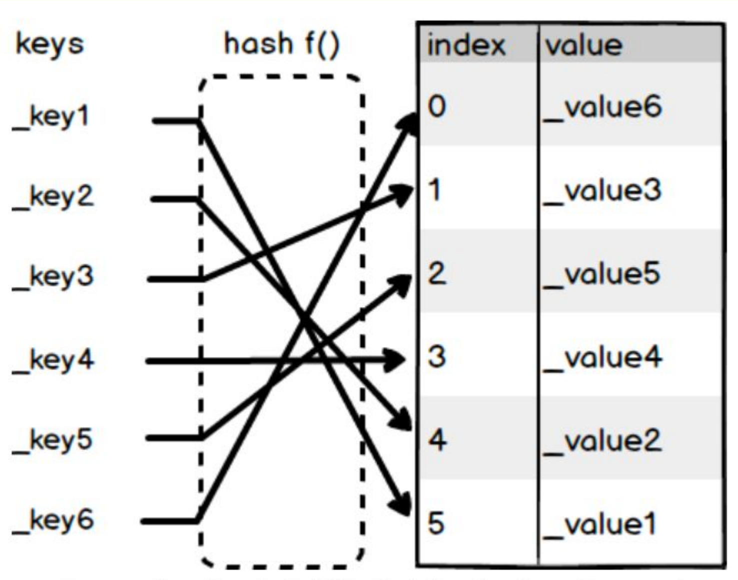
What happens when we go from uint16 to uint8?

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

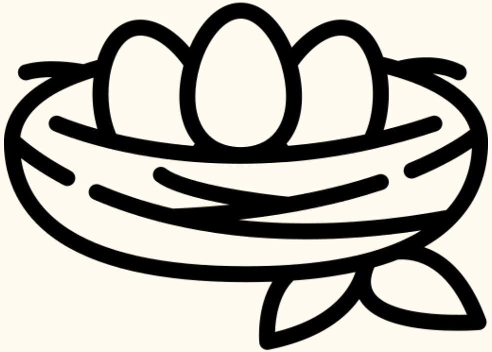
Type - Struct

```
struct Object {

    property1;
    property2;

}
```

```
Object.property
```



Nesting allowed!

Observe tight variable packing

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

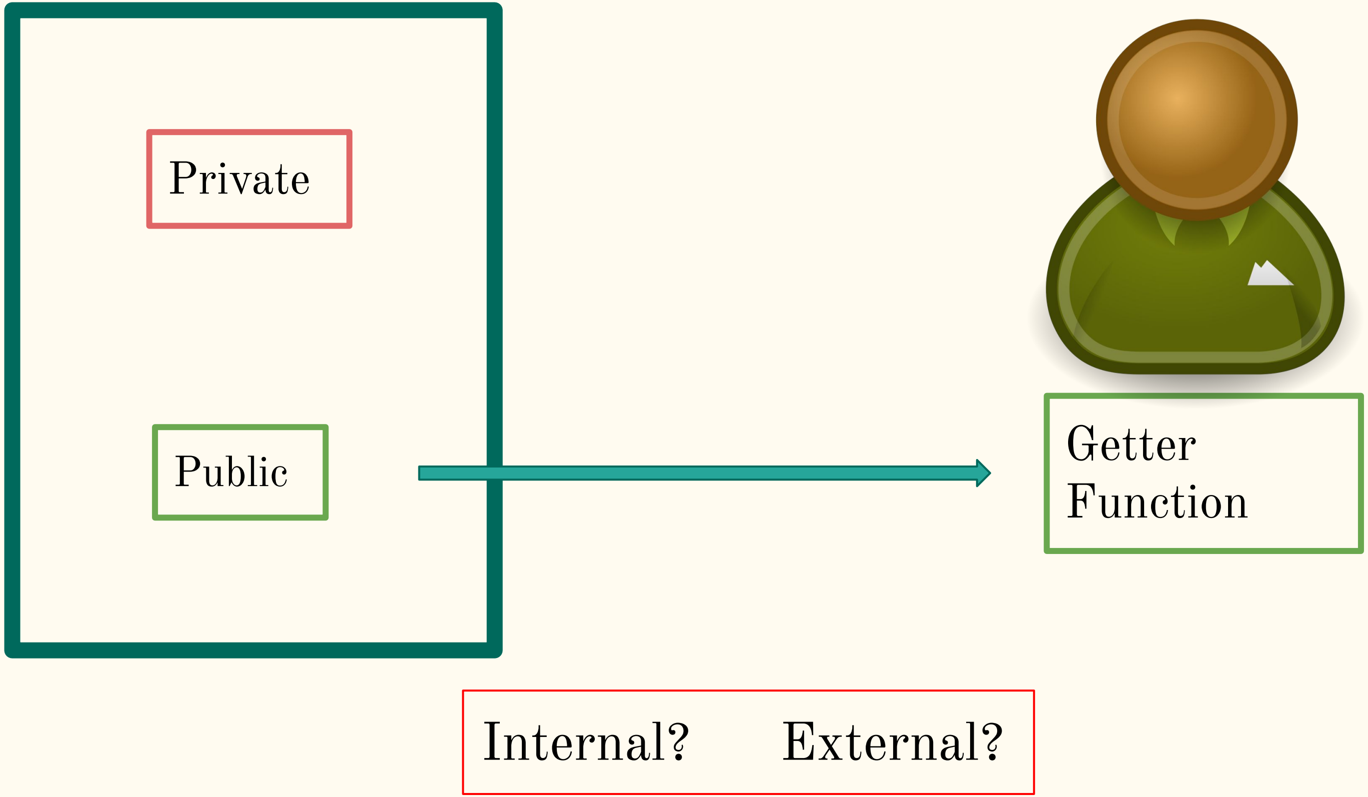
Solidity Variables

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

Type | **Scope** | **Name**

```
mapping(address =>string[]) internal ownerToList
```

```
uint[7] public numbers_fixed;
```

```
struct Person{
        string Name;
        uint8 age;
}
```

```
bool private b;
```

```
uint[] public numbers;
```

```
uint public a;
```

```
Person memory a =
```

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