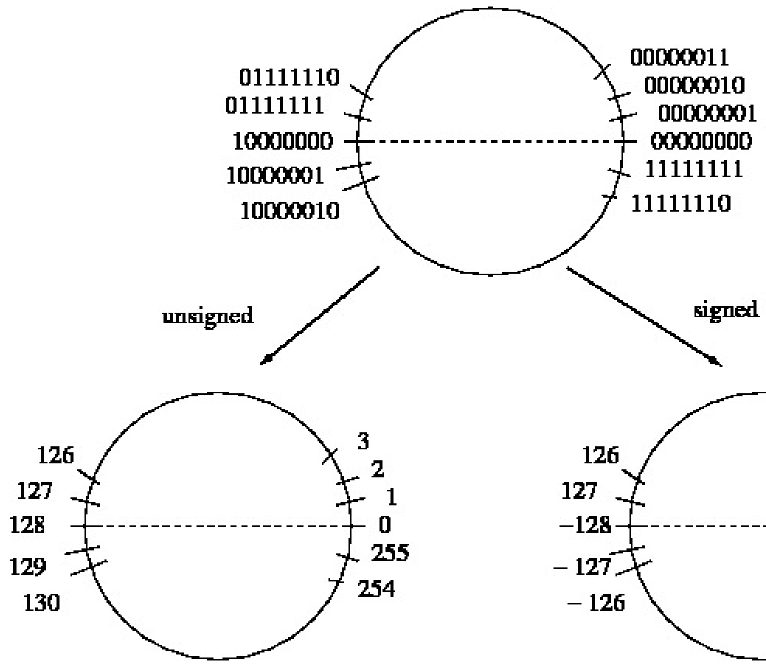
Today I will complete the discussion of binary numbers. Then I will move on and discuss arrays and an algorithm for sorting which uses arrays. The discussion of arrays is a natural lead into the next big topic, lists, which I will begin next lectures.

Unsigned vs. signed numbers

Consider 8 bit numbers (one byte). From what I said last lecture, these numbers would go from 0 to 255. In general, n-bit numbers would go from 0 to $2^n - 1$. Such a representation is called *unsigned* since all the numbers are non-negative. It is illustrated by the figure below on the left.

Note that rather than showing a line of numbers, I have drawn the numbers on a circle. The idea here is that if take $(11111111)_2 = 255$ and you add 1, then you get $(100000000)_2 = 256$ which is has 9 bits rather than 8. If we only keep the "lower" 8 bits, then we have $(00000000)_2 = 0$. Hence, the circle.

To allow for negative numbers, one uses a different interpretation of the binary numbers, which is illustrated in the figure below on the right. This is called the *signed* number representation. Here we count from 0 up to 127 and then, rather than going to 128, we jump to -128. We continue counting up from there to 0. Note that the leftmost bit (the "most significant bit" or MSB) indicates the sign of the number. If the MSB is 0, then the number is non-negative. If the MSB is 1, then the number is negative. Also, note that we are not simply using 7 bits for the number and then using the eight bit for the sign. i.e. (10000001) represents the number -127, not -1.



More generally, the set of *unsigned* n-bit numbers are $\{ 0, 1, 2, \ldots, 2^n - 1 \}$. It is common to use $n = 16, 32, 64$ or 128, though any value of n is possible. The *signed* n-bit numbers are $\{- 2^{n-1}, \ldots, 0, 1, 2, \ldots, 2^{n-1} - 1 \}$.

Here is a table for $n = 8$ and $n = 16$.

binary	signed	unsigned
00000000	0	0
00000001	1	1
:	:	:
01111111	127	127
10000000	-128	128
10000001	-127	129
:	:	:
11111111	-1	255

If n=16, the corresponding table is:

binary	signed	unsigned
0000000000000000	0	0
0000000000000001	1	1
:	:	:
0000000001111111	127	127
0000000010000000	128	128
0000000010000001	129	129
:	:	
0111111111111111	$2^{15} - 1$	$2^{15} - 1$
1000000000000000	-2^{15}	2^{15}
1000000000000001	$-2^{15} + 1$	$2^{15} + 1$
:	:	
1111111101111111	-129	$2^{16} - 129$
1111111110000000	-128	$2^{16} - 128$
1111111110000001	-127	$2^{16} - 127$
:	:	
1111111111111111	-1	$2^{16} - 1$

Funny example from Java

Consider the following lines of Java code:

```
for (short s = 32767; s < 32768; s++)
   System.out.println(s);
```

The number 32767 is $2^{15} - 1$ and so it is the largest `short`. The code is a loop which starts with this number, prints it, and then increments the number. The loop terminates when the value is not less than 32768 which common sense tells you will occur after just one pass through the loop. However, that is not what happens! When the program you write contains the number 32768, Java treats this number as an `int` and this number can indeed be represented correctly as a (32 bit) `int`. When Java adds 1 to the `short` value `s=32767`, it gets -32768. This value is less than 32768 obviously and so the program will print it and indeed it will print all the values from -32768 up to 32767, and then go through that loop infinitely many times. If you don't believe me, try it yourself. *See Exercises 1 Question 10 for a few other examples.*

Unsigned numbers as memory addresses

Binary number representations are used in two ways. The first is to represent *data*. The second is to represent an *addresses* in memory. Addresses are unsigned numbers. When you hear that a computer is a "32 bit machine" this means that there are 2^{32} addressable bytes. A "64 bit machine" may access 2^{64} bytes. Don't be concerned for now about what this actually means in terms of hardware. You'll learn about that in COMP 273. For now, just think of possible addresses of bytes in memory.

As as aside, just to remind you (or in case you didn't know)...

- 2^{10} bytes = 1 kilobyte (1 KB) $\approx 10^3$ bytes (one thousand)
- 2^{20} bytes = 1 megabyte (1 MB) $\approx 10^6$ bytes (one million)
- 2^{30} bytes = 1 gigabyte (1 GB) $\approx 10^9$ bytes (one billion)
- 2^{40} bytes = 1 terabyte (1 TB) $\approx 10^{12}$ bytes (one trillion)
- 2^{50} bytes = 1 petabyte (1 PB) $\approx 10^{15}$ bytes
- 2^{60} bytes = 1 exabyte (1 EB) $\approx 10^{18}$ bytes

The latter two seem like very large numbers. Indeed there are data sets (sometimes called "big data") with that many bytes.

You learned in COMP 202 that Java has primitive types and reference types. Each primitive type in Java uses a certain number of bits (or bytes), namely `boolean, byte, char, short, int, long, float, double` use 1,1,2,2,4,8,4,8, bytes respectively. A variable that is defined as a *primitive type* just stands for a particular set of consecutive bytes of memory where some data is stored.

Variables that have a *reference type* are different. These variables holds the *address* of an object, in particular, the starting address of the object.

You should think of both primitive and reference variables as standing for an address in memory where something is stored. The thing that is stored can either be data itself (in the case of a primitive type), or it could be the starting address of some object (in the case of a reference type).

Note that sometimes I write *starting address* rather than just address. Why? In Java only `boolean` and *byte* types use one byte. Everything else uses multiple bytes. When I say "starting address", I am just reminding you that the item being stored takes more than one byte, and the address only refers to the first of these bytes.

Example

Suppose we define:

```
int  i  = 4;
double  x = 47.35;
```

This defines 4 consecutive bytes somewhere in memory where the integer `i` is stored and 8 consecutive bytes where the double `x` is stored. Whenever your program subsequently has `i` or `x`, it is referring to these two sequences of bytes. We would say that the address of the integer `i` is the first byte of the 4-byte sequence and the address of the double `x` is the first byte of the 8-byte sequence.

Now suppose we have defined a class `Student`, and we have an instruction

```
Student[ ]  studentArray =  new Student[13];
```

This is in fact two declarations

```
 Student[ ]      studentArray;
 studentArray  = new Student[13];
```

The first defines a reference variable which holds the address of an object, namely a `Student` array. Until we construct the object, this address is `null` which is just the number 0. The second line constructs/instantiates a `Student` array which references 13 students. These references are initialized to `null` which is the address 0.

Next we instantiate objects of the class `Student` and use the array to reference them,

```
studentArray[0] = new  Student("Fred");
studentArray[2] = new  Student("Mustafa");
```

In this case, we would have a `Student` object which has some string field in it that is assigned "Fred". This `Student` object would have starting address which is stored the array slot 0 in the `Student` array object.

The key concept to understand here is that any data – whether it is an `int` or a `double` or a an array, or an object or an array of objects – is stored as a set of consecutive bytes in memory. When we talk about the "address" of any particular data item, we are talking about the first of these bytes. We will come back to these ideas throughout the course...