Loops

A loop is a programming structure that repeats the code within the block based on some kind of condition. Each time the loop code block runs we call it an *iteration*.

A condition is a boolean expression that resolves to true or false.

There are two kinds of loops in Java, each with two variants.

while

The while loop runs as long as a condition remains true between each iteration.

```
while( condition )
{
    // code block
}
```

While has a variant called the do-while, in which the code block is executed first before the condition is checked.

```
do
{
    // code block
}
while( condition );
```

When do we use a while loop? A while loop is best used when we do not know exactly how many times we want to iterate something. For example, if we are going to fill a container with tea until it is full, we want to base it not on an absolute amount (since different containers hold different volumes of liquids) but on the capacity of the container. Getting gasoline at a gas station works this way too--the pump runs until the tank is full, unless you've pre-paid and the pump dispenses only a specific amount.

for

The for loop has two variants, one which runs using a *loop control variable* (LCV) and one which does "all items."

```
for( int i = 0; i < data.length; i++ )
{
    // code block
}
```

The standard for loop creates a loop control variable, then increments that variable for each iteration until the condition is no longer true. Because the LCV can represent an index position, it is especially useful for traversing arrays.

```
for( int num : data )
{
    System.out.println( num );
}
```

The for-each loop does not use an LCV, but traverses all elements in a data structure in a linear fashion. It is useful for working with String and ArrayLists where one wants to look at every single object. The statement reads "for each integer num in the list data."

A for loop is best used when one wants to do something a specific number of times.

Recursion

A recursive function is one that calls itself in order to solve a reduced-size version of the exact same problem. Recursion is effective because it allows us to express a problem in a reducible mathematical way.

Each recursive function has a *base case* in which the recursive calls will stop when the solution for the smallest version of the problem is trivial.

That is followed by the *recursive call*, where the problem is reduced in size and the function calls itself to solve it.

We can calculate factorials very easily this way:

```
public static int factorial( int n )
{
    // base case
    if( n == 1 ) return 1;

    // recursive call
    return n * factorial( n - 1 );
}
```

Something more complex like the Fibonacci sequence can be easy to express using recursion, but is not so efficient in terms of computational time:

```
public static int fib( int n )
{
    // base case
    if( n <= 2 ) return 1;

    // recursive call
    return fib( n - 1 ) + fib( n - 2 );
}
```

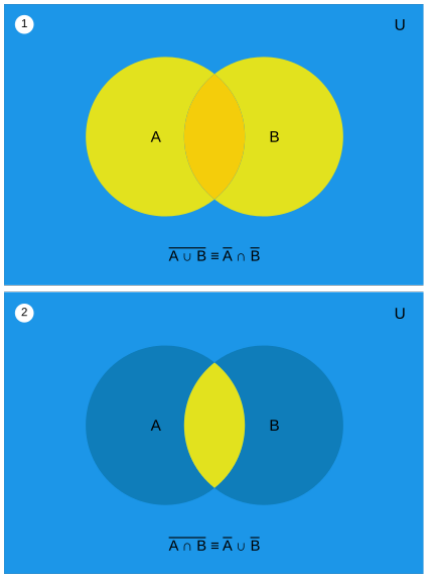
The Fibonacci recursive function here grows exponentially, so the efficiency drops sharply on modern computers once we ask it to calculate past 50 or so numbers in the sequence.

One common question in CS is to ask students to implement this recursive function in an iterative way. In fact, one of the most common problems in CS is to take a recursive function which is easy to understand but inefficient in terms of processor time and resources, and re-implement it in an iterative fashion so that it can run much more efficiently although the code may be more complex and difficult to understand.

A problem we see with testing these is that the integer data type (primitive) is too small to hold the really large integers that we generate. The data type itself will *overflow* and return incorrect values past a certain maximum.

To deal with this, there is a class called BigInteger which uses strings, and can hold any size integer up to the computers available memory.

De Morgan's Laws



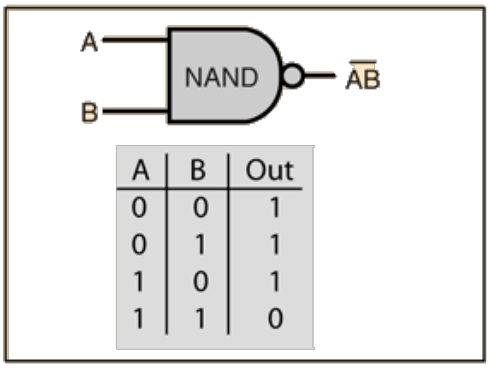
In boolean algebra, these are a pair of transformation rules. In programming and logic gate design, we express them as:

not(A or B) = not A and not B

not(A and B) = not A or not B

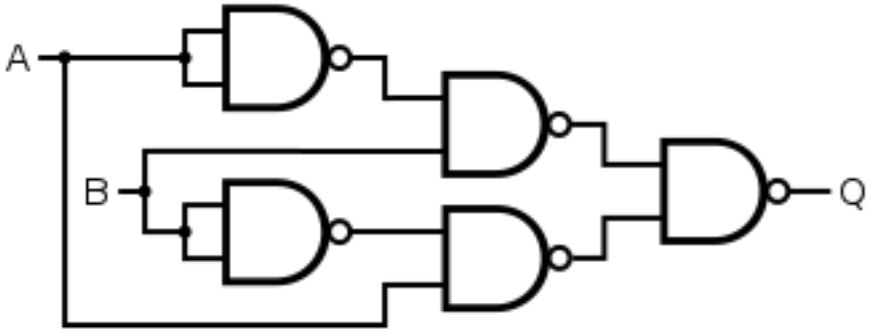
This kind of understanding of logic is what allows us to construct any kind of circuits in hardware using only combinations of NAND ("not and") logic gates.

This is a NAND gate:



A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0

This is an XOR gate made using five NAND gates:
An XOR gate's output is only true when exactly ONE of its inputs is true.



In terms of cost, it is cheaper to make complex gates out of NAND gates (which are cheap to produce) rather than creating all of the specialized ones independently then linking them together.