

Binary Number Project

acknowledgments to Kyle Gillette for this assignment

Welcome to the Binary Number project. In this challenging but interesting exercise, we will explore how decimal numbers are represented in binary form and how to perform mathematical operations using binary numbers. We will also be working more in-depth with `String` operations.

This is a "Pair Programming" Assignment

While working on this project, you will use the "Pair Programming" approach. In pair programming, two programmers share one computer. One student is the "driver," who controls the keyboard. The other is the "navigator," who observes, asks questions, suggests solutions, and thinks about slightly longer-term strategies. I suggest that the 'navigator' has the directions open on their screen, and the 'driver' has the code on their screen. The two programmers should switch roles about every 20 minutes. Working in pairs should make you much better at programming than working alone. The resulting work of pair programming nearly always outshines that of the solitary programmer, with pairs producing better code in less time. For this assignment, each partner will receive the same grade.

You may begin working on this assignment when you have selected your partner and are sitting next to each other. Remember that you must work together as a unified pair to complete this assignment. No 'going rogue' on your own. Good luck, and have fun!

Be sure that both of you have duplicate saved copies of your work each day. If your partner is absent, you need something to work with tomorrow.

Download, extract, and save the Binary Number project to your computer. Open up the project by clicking on the BlueJ package file icon. Notice the (incomplete) `BinaryNumber` class and the `BinaryNumberTest` class (listed as 'no source'). You may run `BinaryNumberTest` at any time to see how much of the assignment you have completed.

Your assignment is to complete the `BinaryNumber` class that represents (guess what?) binary numbers. A binary number is a number expressed in the base-2 (binary) numeral system, a method of mathematical expression that uses only two symbols: typically "0" (zero) and "1" (one). Whereas the digits in a decimal number (from right to left) represent increasing powers of 10, the digits in a binary number (from right to left) represent increasing powers of 2. Here is a table showing the digit representation of the equivalence between decimal and binary numbers.

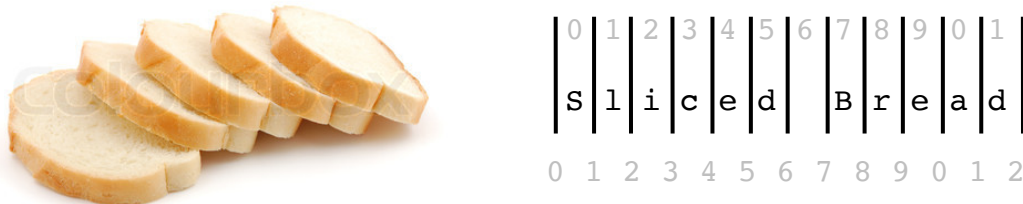
Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Binary	0	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111	10000

Binary numbers are useful because each digit (or *bit*) can represent an electrical circuit's on or off state, which enables the use of transistor semiconductors (electronic switches). As a result, almost all modern computers and computer-based devices use the binary system.

For this project, the `BinaryNumber` class will represent a binary number as a `String` composed of 1s and 0s. (For example, "10011001".) While the length (number of digits or bits) of a `BinaryNumber` is arbitrary, you should assume that all `BinaryNumbers` used for calculations will all have the same number of bits (digits).

Here are the instructions for you to complete the `BinaryNumber` class:

1. Write a constructor that accepts a `String` as a parameter and uses it to set the instance variable.
2. Complete the `getNumber` method that returns the number (a `String`).
3. Now the assignment gets a bit (ha!) more challenging. Do you recall how the `substring` method of the `String` class works? There are two versions of it (it is an *overloaded* method), namely `substring(beginIndex)` and `substring(beginIndex, endIndex)`. A helpful way to remember how either version of `substring` works is to think of a loaf of sliced bread, as in the following example.



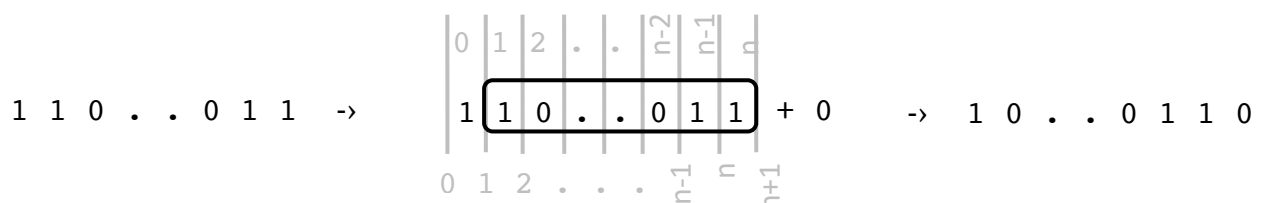
Think of the slices (as numbered on the bottom) as isolating a specific piece of bread (as numbered on the top).

For example (given the diagram above), if you were looking for everything except the *first* letter, you would start at slice 1 and take the rest of the loaf. In the same way, `substring(1)` would yield (in this case) the `String` "liced Bread" (Yuk! That's disgusting!).

For another example (given the diagram above), if you were looking for everything except the *last* letter, you would start at slice 0 and go through slice 11 (or $n - 1$ for a string that is n characters in length). In the same way, `substring(0, 11)` would yield (in this case) the `String` "Sliced Brea" (Now that sounds better!).

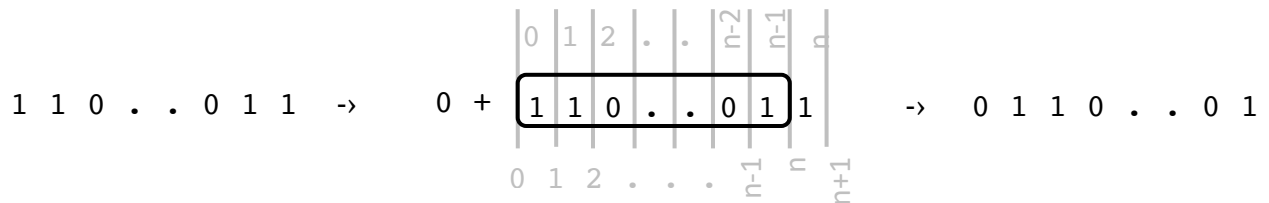
As a third example (given the diagram above), if you were looking for the letter in position 3 (the "c" in "sliced"), you would start with slice number 3 and end with slice number 4. In the same way, a call to `substring(3, 4)` would yield the single character `String` "c".

Now, we will put that knowledge to use in writing the next three methods. First, complete the `shiftLeft()` method that returns a new `BinaryNumber`, where each bit is moved one position to the left, eliminating the left-most bit, and the right-most bit automatically becomes a zero. Note that the number of digits in the new and old binary numbers remains the same. You may find it helpful to remember that `substring(beginIndex)` returns the `String` starting at `beginIndex` through the rest of the `String`. Find the proper substring, concatenate a 0 to the end of it, and use it to return `new BinaryNumber(String);`.



Notice that because each bit moves over one position to the left, this method creates a new `BinaryNumber` twice the original value (hence this method performs multiplication by 2). Can you see the inherent problem with possible mis-evaluation because of the discarded left-most digit?

- Next, complete the `shiftRight()` method that returns a new `BinaryNumber` where each bit is moved to the right, the right-most bit is eliminated, and the left-most bit automatically becomes a zero. Note that the number of digits in the new and old binary numbers remains the same. You may find it helpful to remember that `substring(beginIndex, endIndex)` returns the `String` starting at `beginIndex` and ending at `endIndex - 1`.



Notice that because each bit moves over one position to the right, this method creates a new `BinaryNumber` that is half the value of the original (hence this method performs division by 2). Can you see the inherent problem with loss of precision because of the discarded right-most digit? This is the issue with integer division as the 'one's digit' is always ignored, so odd numbers always divide evenly. For example, when `0111` (which is 7) is halved (shifted right), it becomes `0011` (which is 3).

- Your next task is to complete the `getBit` method that looks at the bit in the given position (starting from the right-hand side at position 0) and returns `true` if the bit is a 1 or `false` if the bit is a 0. For example, if the binary number were `"00001000"`, a call to `getBit(3)` would return `true`, while a call to `getBit(4)` would return `false`. You'll have to write a routine that will look up the single character at the given location and then return either `true` or `false` based on what it is. There are several ways to do this, but probably the most obvious is to use the `String` class's `substring(beginIndex, endIndex)` method.

Remember that for this method, however, the `String` positions start at 0 on the right-hand side and go up to the left. You will have to subtract to figure out the appropriate opposite position. It might help you to remember the `String` class's `length()` method, which returns the number of characters in the entire string (in other words, its length). Perhaps you have already realized that the calculation to find the proper start position is:

```
number.length() - position - 1
```

Another important consideration is the proper way to compare two objects for equality, particularly two `String` objects. **Never test for `String` equality like this: `oneString == anotherString`.** The 'double equals sign' (`==`) compares if the two `String` objects have *the same address in memory*, not if they contain the same characters. You should use the `String` class's built-in `equals` method instead, specifically designed to compare the characters in each string to see if they are the same. The proper way to see if the two `Strings` are equivalent is to call `oneString.equals(anotherString)`. We'll learn later that properly written objects

that will be compared to one another need to include an `equals` method that defines how equality is to be measured.

Make sure the `getBit(int position)` method is working correctly before going further. All of the other remaining methods are dependent on this one working properly.

6. Write a `not()` method that returns a new `BinaryNumber` with the opposite values to the original `BinaryNumber`. In other words, any 1s become 0s, and any 0s become 1s. The number of digits remains the same.

You will probably need to loop through each bit, one at a time (using `getBit`), and then build a new `String` with the opposite values. Be careful to think about how you are looping through the original binary number string, and then consider the order of the digits as you build the new binary number string. It's easy to make a mistake by accidentally creating a new string in reverse order.

7. Complete the `and` method that takes another `BinaryNumber` as a parameter and returns a new `BinaryNumber`. The two `BinaryNumbers` used to form a third new one are the existing `BinaryNumber` (this, the one you are in right now) and the other `BinaryNumber` (other) passed as a parameter. Loop through each of those two `BinaryNumbers` looking at the same bit position in each of them. Since both binary numbers have the same number of digits, you can use a single loop and then use the loop index to get the appropriate bit from both numbers. The rules for the `and` method come directly from Boolean logic; if both bits are `true`, then that same bit in the new (third) binary number is set to `true`. If either of the two bits is `false`, that bit in the new binary number is `false`. You get the idea.

111000
<u>and 100110</u>
100000

8. Complete the `or` method that takes another `BinaryNumber` as a parameter and returns a new `BinaryNumber`. The `or` method is very similar to the `and` method you just wrote, except that if *either* of the two bits examined is `true`, the corresponding bit in the new binary number is `true`. The only time the new bit will be `false` is if both bits compared are `false`.

111000
<u>or 100110</u>
111110

9. Complete the `xor` (known as an *exclusive or*) method that takes another `BinaryNumber` as a parameter and returns a new `BinaryNumber`. The `xor` method is very similar to the `or` method you just wrote, except that if both bits are different (one `true` and one `false`), then the corresponding bit in the new binary number will be `true`. Conversely, if both bits are the same, then the corresponding bit in the new binary number will be `false`.

111000
<u>xor 100110</u>
011110

10. Complete the `add` method that takes another `BinaryNumber` as a parameter and returns a new `BinaryNumber`. While this method uses the same looping procedure you did for the previous three methods, this one requires careful consideration. Just like adding two decimal numbers, if the sum of two digits is more than 9, you must keep track of the 'carry' number that spills over into the next digit's calculation. The same thing holds true for binary numbers (which, of course, can only be 1s or 0s). In other words, when adding two binary bits together, you also have to consider the amount (if any) that was 'carried over' from the calculation of previous digits. So there are essentially three different bit values that have to be added together for any given digit

1111
011110
<u>add 000111</u>
100101

calculation, one bit from `this` binary number, one bit from the `other` binary number, and the 'carry' bit (if any). The possible sum of those three bit values is 0, 1, 2, or 3. A digit total of 2 or 3 means that 2 should be subtracted from the total, and the 'carry' bit should be set to 1. Suppose you have an integer variable called `count` that stores the given digit total, an integer variable named `carry` that stores the digit carry value, and a `String` `newNumber` used to create the new `BinaryNumber`. In that case, as you loop through each digit position starting at 0, the following routine might prove helpful to you:

```
int count = 0;
if(this.getBit(i))
    count++;
if(other.getBit(i))
    count++;
if(count > 1)
{
    count--;
    carry++;
}
newNumber = (count % 2) + newNumber;
carry += count / 2;
```

11. And now, for the most intriguing part of this assignment, let us consider the mathematical numbering concept known as *two's complement* in detail. A two's-complement number encoding system allows binary numbers to represent both positive and negative values. Specifically, two's complement is the additive inverse of a binary number. In other words, when a binary number and its "two's complement" are added together, the sum is always zero. For example, given the equation $x + y = 0$, solving for x yields $x = -y$. This is how calculating two's complement can be used to represent a negative value using binary numbers.

So how do we calculate a two's complement binary value? It is much easier than it sounds. Simply take the inverse (using the `not` method) of the binary number and then add 1 to it.

Note that simply adding a binary number to its opposite (`not`) does not equal zero (actually, this is the *one's complement*). Rather the sum will be a binary number consisting of all 1s, which in a standard binary numbering scheme represents the highest numerical value that a binary number of a given length can store. Ah, but if you add 1 to that total, all of the 1s flip to 0s, *except* for an extra last leading overflow 1s digit that comes from the carry forward from all the previous digits. But remember, the number of digits of a binary number does not ever change (the number of bits is constant) so that extra overflow leading digit is simply thrown away, and we wind up with all 0s instead.

	011110
<u>add (not)</u>	<u>100001</u>
	111111
	111111
<u>add (one)</u>	<u>000001</u>
	1000000

	111111
	011110
<u>add (2's)</u>	<u>100010</u>
	1000000

Complete the `twosComplement` method that returns a new `BinaryNumber`. Use the `not` method to create a new `BinaryNumber` opposite the original, then add 1. Note that to add 1 to any binary number correctly, you need to figure out how many digits are in the binary number you are adding it to, then fill in the appropriate amount of leading zeros of your binary number 1.

Start with a `String` literal "1", and fill in the proper amount of zeros on the left side. How many zeros, you ask? It's the `length()` of the binary number minus one.

12. Write the `binary2decimal` method that returns the decimal (base 10) integer value of the binary number. For example, the binary number 00110011 is equivalent to the decimal number 51. How do you convert a binary number to a decimal number, you ask? Start at bit position 0; if the bit is 1 (true), add $2^{(\text{position})}$ to the total.

As described above, our `BinaryNumbers` class will follow the two's complement encoding scheme widely used (including by Java) for representing negative binary numbers. Two's complement uses the most significant bit (the last one on the left) as a 'sign bit'. Using a 'sign bit', we get a new range of numbers available – specifically negative numbers. Therefore, if the last bit (the one on the far left) is true, then the number is negative. Check the last bit once more (after you have summed up all the values); if it is `true`, subtract $2^{(\text{number of digits})}$ from the total. For example, usually the 6-bit binary number 111111 would be equivalent to 63 but using a two's complement encoding scheme, it would be $63 - 2^6$ (or 64), which equals -1. A handy check is that *any two's complement binary number composed of 1s is always equivalent to -1*.

So, think about it for a minute; if we have an 8-bit number (one byte), we can generally represent 0 – 255 (which represents 2^8 numbers). Does that mean that by using the most significant bit (the one on the far left representing 2^7) as a 'sign bit', we can represent all the values between negative 255 and positive 255? Unfortunately, no, because binary and decimal numbers have a 1-to-1 relationship, and if 8 bits can only represent 256 different binary numbers, it also can only represent 256 unique decimals. Because that last (and most significant) bit determines the sign, the largest positive number that can be stored in 8 bits is 01111111 (which is 127), while the smallest negative number is 10000000 (which is $128 - 2^8$ [or 256] which equals -128). Thus, we get the range from -128 to 127, which (including 0, of course) is 512 different numbers.

Then why don't we just keep a separate bit dedicated to the number sign and use the rest for number values, you ask? (By the way, this is called a *sign and magnitude* encoding scheme.) Using this scheme, the 8-bit binary number range would go from 0 to 127 (based on the first 7 bits) and either positive or negative (based on the last bit), which would include both a positive and a negative zero (both of which are literally nonsensical). One's complement (which is simply flipping all the 1s to 0s and 0s to 1s) does the same thing (there are two values for zero). Using two's complement, on the other hand, gives only one zero representation and an extra number value to boot. That's why it is so widely used.

Say we have the maximum positive value for a 10-bit two's complement binary number, which would be 0111111111, or 511 in decimal form. What happens when we add 1 to that number? In decimal form, the answer is obviously 512, but it is not so apparent in a two's complement (or any form of) binary number representation.

decimal	binary	2's Comp
511	0111111111	(511)
+ 1	+ 0000000001	+ (1)
512	1000000000	(-512)

This resulting miscalculation is called (can you guess?) arithmetic overflow. It happens when we try to represent a number outside the range of maximum (or minimum) values. We don't encounter this problem with decimal values *because we simply add another significant leading*

digit if needed. You can't do that with binary numbers because their number of bits is fixed from the beginning, so their range is automatically limited. Why don't we just make them with lots of bits from the start? Because every binary number would have that large number of bits, which would use up a significant amount of computer resources (both memory and speed). In Java, a primitive `int` is represented by 4 bytes (32 bits) and uses a two's complement encoding scheme. That means that an `int` can represent from -2,147,483,648 to 2,147,483,647 (from -2^{31} to $2^{31} - 1$), and if you try to add 1 to the maximum integer value, you will 'wrap around' and get the minimum integer value. This is the source of arithmetic overflow errors.

If you are interested, there is a simple test using a two's complement encoding scheme for detecting when arithmetic overflow occurs when adding two numbers. You can be sure that arithmetic overflow has occurred if two positive numbers added together yield a negative number, and conversely if two negative numbers added together yield a positive number.

Looping through each binary digit (from right to left), here is the two's complement algorithm for calculating the decimal value of a binary number.

```
if (getBit(i))
{
    decimal += Math.pow(2, i);
    if (i == number.length() - 1)    // sign bit
        decimal -= Math.pow(2, i + 1);
}
```

13. Lastly, complete the `toString` method that includes both the binary and decimal representations of the number. For example, your `toString` might return a `String` that contains something like "Binary: 00011 Decimal: 3". The exact format (letters, spacing, etc.) is not essential, just so long as it contains both the binary and decimal values of the number.

Good luck. Show me the results of your successful test when you are finished. Feel free to see me if you have any questions or difficulties.