Logic Gates and Java

By: Ethan J. Nephew

Introduction

The purpose of this document is to examine logic gates to a non-superficial depth and to explore how they manifest within the Java language. There are seven basic logic gates: AND, OR, NAND, NOR, NOT, XNOR, and XOR. The AND, OR, and NOT operators are typically the most familiar or prolific within Java programming.

Logic Gates With Operators

OR	0	1
0	0	1
1	1	1

Figure 1 If one is provided as any of the inputs to an OR gate, then it will resolve to one.

The **OR** gate, figure 1, only evaluates to 0 when all inputs are 0. If any of the inputs are a 1, then the output evaluates to a 1. An OR gate is appropriately applied when it is necessary to check for any degree of truth. Java operator for OR is **||**. For numerical evaluation a single **|** is used.

- 1 || 1 = 1
- 1 || 0 = 1
- 0 || 0 = 0
- 0 || 1 = 1

AND	0	1
0	0	0
1	0	1

Figure 2 AND will only resolve to one if all inputs are one.

The **AND** gate, figure 2, only evaluates to 1 if all the inputs are 1. AND is a check that only passes if all inputs are true or 1. This gate can be viewed as a concrete check. It only resolves to true or 1 when all items being compared are true or 1. The Java operator for AND is **&&**. For numerical evaluation a single **&** is used.

- 1 && 1 = 1
- 1 && 0 = 0
- 0 && 0 = 0
- 0 && 1 = 0

NOT	
0	1
1	0

Figure 3 NOT resolves to the opposite of whatever is provided to it.

The **NOT** gate, figure 3, evaluates to the opposite of whatever is provided. If true, then false. If false, then true. If 1, then 0. If 0, then 1. Often, the NOT gate is used as a quick check to validate a piece of data. It is typical to see NOT added to a Boolean expression. The Java operator for NOT is **!**. For numerical evaluation **~** is used.

- !1 = 0
- !0 = 1

XOR	0	1
0	0	1
1	1	0

Figure 4 XOR is a defined operator in Java. It returns 1 only when there is a divergence in its inputs.

The **XOR** gate, figure 4, checks for a divergence among the comparators. 1 is only returned when the values being compared are different from one another. The Java operator for XOR is **^**.

- 1 ^ 1 = 0
- 1 ^ 0 = 1
- 0 ^ 0 = 0
- 0 ^ 1 = 1

Logic Gates Without Operators

The NOR gate is the inverse of the OR gate. Rather than returning 1 for any equation where 1 is an input, the NOR gate will only return 1 if both inputs are 0.

- 1 NOR 1 = 0
- 1 NOR 0 = 0
- 0 NOR 0 = 1
- 0 NOR 1 = 0

NOR	0	1
0	1	0
1	0	0

Figure 5 The NOR gate only returns 1 if all items being compared are 0.

The NAND gate can be considered the opposite of the AND gate. Where the AND gate only returns 1 when both inputs are 1, the NAND gate only returns 0 when both inputs are 1.

- 1 NAND 1 = 0
- 1 NAND 0 = 1
- 0 NAND 0 = 1
- 0 NAND 1 = 1

NAND	0	1
0	1	1
1	1	0

Figure 6 The NAND gate can be viewed as the inverse output of an AND gate.

The XNOR gate can be considered the opposite of the XOR gate. Where the XOR gate would return a 1, the XNOR gate will return a 0. Where the XOR gate would return a 0, the XNOR gate will return a 1.

- 1 XNOR 1 = 1
- 1 XNOR 0 = 0
- 0 XNOR 0 = 0
- 0 XNOR 1 = 1

XNOR	0	1
0	1	0
1	0	1

Figure 7 The XNOR gate is the inverse of the XOR gate.

Java Booleans and Logic Gates

```
public static void main(String[] args){
    boolean True = true, False = false;
    System.out.println(True && True); // true AND true evaluates to true
    System.out.println(True && False); // true AND false evaluates to false
    System.out.println(False && True); // false AND true evaluates to false
    System.out.println(False && False); // false AND false evaluates to false
}
```

Figure 8 This is an example in Java of how AND can be used.

```
public static void main(String[] args){
    boolean True = true, False = false;
    System.out.println(True || True); // true OR true evaluates to true
    System.out.println(True || False); // true OR false evaluates to true
    System.out.println(False || True); // false OR true evaluates to true
    System.out.println(False || False); // false OR false evaluates to false
}
```

Figure 9 This is an example in Java of how OR can be used.

```
public static void main(String[] args){
    boolean True = true, False = false;
    System.out.println(!True); // NOT true evaluates to false
    System.out.println(!False); // NOT false evaluates to true
}
```

Figure 10 This is an example in Java of how NOT can be used.

```
public static void main(String[] args){
    boolean True = true, False = false;
    System.out.println(True ^ True); // true XOR true evaluates to false
    System.out.println(True ^ False); // true XOR false evaluates to true
    System.out.println(False ^ True); // false XOR true evaluates to true
    System.out.println(False ^ False); // false XOR false evaluates to false
}
```

Figure 11 This is an example, in Java, that shows the behavior of the XOR operator as specified above in Figure 6.

```
public static void main(String[] args){
    boolean True = true, False = false;
    System.out.println(!(True && True)); // true NAND true evaluates to false
    System.out.println(!(True && False)); // true NAND false evaluates to true
    System.out.println(!(False && True)); // false NAND true evaluates to true
    System.out.println(!(False && False)); // false NAND false evaluates to true
}
```

Figure 12 NAND does not have an operator in Java, but it can be represented by using !(a && b).

```
public static void main(String[] args){
    boolean True = true, False = false;
    System.out.println(!True && !True); // true NOR true evaluates to false
    System.out.println(!True && !False); // true NOR false evaluates to false
    System.out.println(!False && !True); // false NOR true evaluates to false
    System.out.println(!False && !False); // false NOR false evaluates to true
}
```

Figure 13 NOR does not have an operator in Java, but it can be represented by using (!a && !b).

```
public static void main(String[] args){
    boolean True = true, False = false;
    System.out.println(!(True ^ True)); // true XNOR true evaluates to true
    System.out.println(!(True ^ False)); // true XNOR false evaluates to false
    System.out.println(!(False ^ True)); // false XNOR true evaluates to false
    System.out.println(!(False ^ False)); // false XNOR false evaluates to true
}
```

Figure 14 XNOR does not have an operator in Java, but it can be represented by using !(a ^ b).

Logic Gates and Numbers

In Java, logic gates are often used in a Boolean context, however, they can be used in numbers too. They do not share the same syntax as the Boolean checks, but they are recognizable.

AND, &, and &&

```
System.out.println(40 && 10);
```

Operator '&&' cannot be applied to 'int', 'int'

Figure 15 It can be observed that the IDE shows an error when applying the && operator on numbers. This occurs when attempting to apply the || operator on numbers too, because && and || are Boolean operators in Java. For comparing numbers use & or |.

Instead of using the && operator, with numbers it is necessary to use the & operator. This operator will check numbers and return a number that is a result of the operation.

```
System.out.println(40 & 10); // Prints 8
```

Figure 16 This is how AND can be applied to a number. Why does this print 8?

The result of 40 & 10 is 8? At first glance, this seems like a strange output. It appears strange because this is the resulting number that is calculated when applying the AND gate to the decimal numbers in binary.

		1 AND 40 is 8							
		128	64	32	16	8	4	2	1
	40	0	0	1	0	1	0	0	0
&	10	0	0	0	0	1	0	1	0
	8	0	0	0	0	1	0	0	0

AND	0	1
0	0	0
1	0	1

Figure 17 The operation checks each number in binary against each other. In the AND gate, 1 is only returned when both inputs are 1. In this case, both inputs are only 1 on the 8 in binary. Therefore, the decimal 8 is the result.

OR, |, and ||

OR can be used with numbers, in much the same way as AND is. Rather than using the || operator, instead, the | operator is used. Just like the & operator the | operator compares the inputs in binary and returns the result in decimal. At this point, it should be possible to predict what 40 | 10 is.

		40 | 10 is 42							
		128	64	32	16	8	4	2	1
	40	0	0	1	0	1	0	0	0
|	10	0	0	0	0	1	0	1	0
	42	0	0	1	0	1	0	1	0

OR	0	1
0	0	1
1	1	1

Figure 18 The OR operation compares each decimal number in binary. The OR logic will only return 0 if both inputs are 0.

NOT, !, and ~

NOT is a little more complicated because it uses the 2's complements. This is similar to regular binary conversions; however, negative numbers are accounted for. In a 2's complement, the furthest left binary is meant to signify a negative number. All the binary columns to the right still represent positives. The negative binary is then calculated by performing addition.

Positive				
8	4	2	1	
0	0	0	0	**0**
0	0	0	1	**1**
0	0	1	0	**2**
0	0	1	1	**3**
0	1	1	0	**4**
0	1	0	1	**5**
0	1	1	0	**6**
0	1	1	1	**7**
1	0	0	0	**8**

Negative				
-8	4	2	1	
0	0	0	0	**0**
1	1	1	1	**-1**
1	1	1	0	**-2**
1	1	0	1	**-3**
1	1	0	0	**-4**
1	0	1	1	**-5**
1	0	1	0	**-6**
1	0	0	1	**-7**
1	0	0	0	**-8**

- **-8+4+2+1 = -1**
- **-8+4+2+0 = -2**
- **-8+4+0+1 = -3**
- **-8+4+0+0 = -4**
- **-8+0+2+1 = -5**
- **-8+0+2+0 = -6**
- **-8+0+0+1 = -7**
- **-8+0+0+0 = -8**

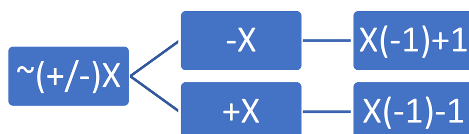
Figure 19 In the Negative table, observe how the calculations are the simple result of adding up all the number columns that contain a 1.

When the NOT operation begins, the binary comparison goes one column beyond the provided number. This results in the sign of the number (positive or negative) being flipped. So, when a positive number is provided in a NOT calculation, the result will be negative. When a negative number is provided in a NOT calculation, the result will be positive. Careful examination of figure 19 should provide sufficient information to understand negative calculations in binary.

		NOT 6 is ?				
		(+/-) 8	4	2	1	
~	6	0	1	1	0	Pos
	-7	1	0	0	1	Neg

Figure 20 This is the result of the NOT calculation. Review Figure 19 for the red row.

The calculation begins with the provided number in binary, 6, is 1 0 1. To flip the sign, it then goes one beyond, so it is comparing 0 1 1 0. After that, it flips each number. 0 1 1 0 becomes 1 0 0 1, then the far-left binary is applied as a negative. Therefore, the resulting calculation is -8 + 0 + 0 + 1 = -7. At first glance, it does seem a little confusing, but understanding how negative numbers in binary are represented clears up the confusion.



For conducting simple NOT calculations the formula to the left can be used. The branches present an input that is positively signed and/or an input that is negatively signed.

XOR, and ^

XOR is the last operator in Java that can be used on numbers. XOR is calculated in much the same way as OR and AND. Decimal numbers are compared on a binary level and the XOR logic is applied to each binary column.

	40 ^ 10 is 34							
	128	64	32	16	8	4	2	1
40	0	0	1	0	1	0	0	0
⊕ 10	0	0	0	0	1	0	1	0
34	0	0	1	0	0	0	1	0

XOR	0	1
0	0	1
1	1	0

Figure 21 The XOR operator accesses the decimal number in binary and applies the logic gate to each binary column.

Application of XOR with Numbers

One interesting way that the XOR operator can be applied is for finding a single missing number in a sequence of non-repeating numbers. For example, if provided {1,2,4,5} the XOR operator can be used to determine that 3 is the missing number. Additionally, the sequence can be unsorted, {5,1,2,4}, so long as there is a single number that is missing.

int[] arr = {1,3}

a1 ^ a2 ^ a3 ^ …^ an-1 = a

	1 XOR 3 is 2							
	128	64	32	16	8	4	2	1
1	0	0	0	0	0	0	0	1
⊕ 3	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0

a1 ^ a2 ^ a3 ^ …^ an = b

	1 XOR 2 is 3							
	128	64	32	16	8	4	2	1
1	0	0	0	0	0	0	0	1
⊕ 2	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	1	1

	3 XOR 3 is 0							
	128	64	32	16	8	4	2	1
3	0	0	0	0	0	0	1	1
⊕ 3	0	0	0	0	0	0	1	1
0	0	0	0	0	0	0	0	0

a ^ b = an

	1 XOR 3 is 2							
	128	64	32	16	8	4	2	1
1	0	0	0	0	0	0	0	1
⊕ 3	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0

Figure 22 This is an example of how the equation can be applied. 2 is the missing number.

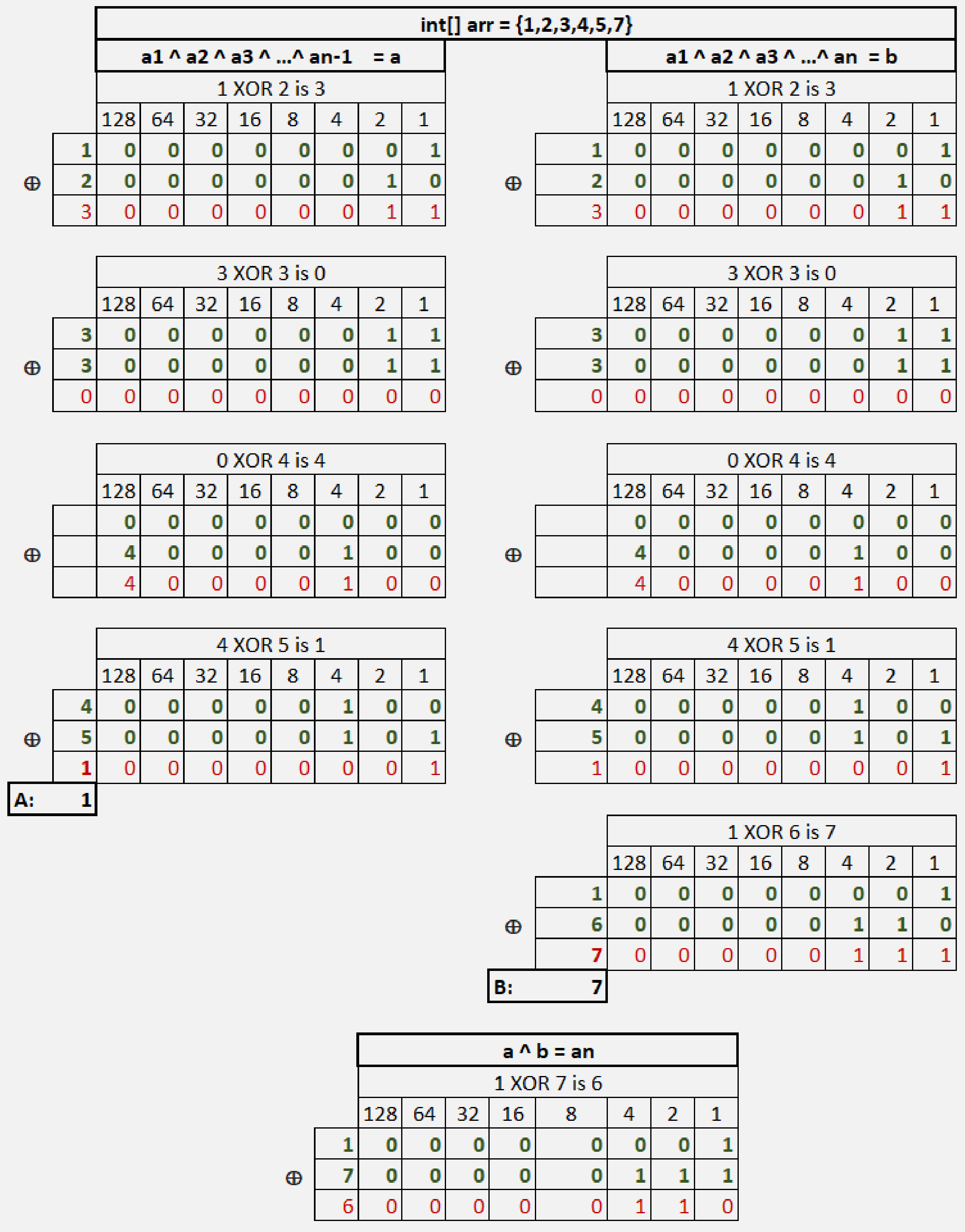
This method of calculating a missing number is derived from two fundamental principles:

1. X ^ X = 0
2. There is a missing number in the array.

Since we know that performing the XOR operation on two instances of the same number will equal 0, then we can effectively cancel out the value that would be same number. If a number is missing, then its value will persist throughout the equation.

As an analogy, the equation can be viewed as a type of scale. The missing number represents an imbalance on the scale. The true value of the imbalance remains cloaked within a series of XOR operations. The final XOR calculation is when the missing value is revealed.

If we know a number is missing, then we can say that the true size of the array should be n+1 or the given array plus the missing number. Iterating through the given array and applying the XOR operator will produce a number. If no number in the sequence is missing, then the next number that should exist in the sequence will be produced. However, if a number is missing, then the number produced will be that missing number.



int[] arr = {1,2,3,4,5,7}

a1 ^ a2 ^ a3 ^ ...^ an-1 = a

1 XOR 2 is 3

		128	64	32	16	8	4	2	1
	1	0	0	0	0	0	0	0	1
⊕	2	0	0	0	0	0	0	1	0
	3	0	0	0	0	0	0	1	1

3 XOR 3 is 0

		128	64	32	16	8	4	2	1
	3	0	0	0	0	0	0	1	1
⊕	3	0	0	0	0	0	0	1	1
	0	0	0	0	0	0	0	0	0

0 XOR 4 is 4

		128	64	32	16	8	4	2	1
		0	0	0	0	0	0	0	0
⊕		4	0	0	0	0	1	0	0
		4	0	0	0	0	1	0	0

4 XOR 5 is 1

		128	64	32	16	8	4	2	1
	4	0	0	0	0	0	1	0	0
⊕	5	0	0	0	0	0	1	0	1
	1	0	0	0	0	0	0	0	1

A: 1

a1 ^ a2 ^ a3 ^ ...^ an = b

1 XOR 2 is 3

		128	64	32	16	8	4	2	1
	1	0	0	0	0	0	0	0	1
⊕	2	0	0	0	0	0	0	1	0
	3	0	0	0	0	0	0	1	1

3 XOR 3 is 0

		128	64	32	16	8	4	2	1
	3	0	0	0	0	0	0	1	1
⊕	3	0	0	0	0	0	0	1	1
	0	0	0	0	0	0	0	0	0

0 XOR 4 is 4

		128	64	32	16	8	4	2	1
		0	0	0	0	0	0	0	0
⊕		4	0	0	0	0	1	0	0
		4	0	0	0	0	1	0	0

4 XOR 5 is 1

		128	64	32	16	8	4	2	1
	4	0	0	0	0	0	1	0	0
⊕	5	0	0	0	0	0	1	0	1
	1	0	0	0	0	0	0	0	1

1 XOR 6 is 7

		128	64	32	16	8	4	2	1
	1	0	0	0	0	0	0	0	1
⊕	6	0	0	0	0	0	1	1	0
	7	0	0	0	0	0	1	1	1

B: 7

a ^ b = an

1 XOR 7 is 6

		128	64	32	16	8	4	2	1
	1	0	0	0	0	0	0	0	1
⊕	7	0	0	0	0	0	1	1	1
	6	0	0	0	0	0	1	1	0

Figure 23 This is an example of how the XOR operator can be used to calculate the missing value.