Short Circuit Evaluation of Java's Boolean Operators

Here's a table describing four of Java's boolean operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Short circuit?</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>and</td>
<td>yes</td>
</tr>
<tr>
<td>&amp;</td>
<td>and</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>or</td>
</tr>
</tbody>
</table>

The && and || operators are short circuit operators. A short circuit operator is one that doesn't necessarily evaluate all of its operands. Take, for example, the operator &&. What happens when Java executes the following code?

```java
if (0 == 1 && 2 + 2 == 4) {
    out.println("This line won't be printed.");
}
```

You might expect Java to ask itself if 0 equals 1, and then ask if 2 + 2 equals 4. But with Java's && operator, that's not what happens. Instead, Java does the following:

Evaluate 0 == 1, discovering that 0 == 1 is false.

Realize that the condition (0 == 1 && whatever) can't possibly be true, no matter what the whatever condition happens to be.

Return false (without bothering to check if 2 + 2 == 4).

The condition (0 == 1 && whatever) has to be false, because 0 == 1 is false. (Remember, the && operator wants both conditions, on its left and right sides, to be true.)

So when Java finds the value on the left side of an && operator to be false, then Java gives up and declares the entire expression to be false. That's called short circuit expression evaluation. The same kind of thing happens with the || operator (another short circuit operator) when the value on the operator's left side is true.

```java
if (2 + 2 == 4 || 0 == 1) {
    out.println("This line will be printed.");
}
```

Here's how Java's || operator behaves when it encounters this code:

Evaluate 2 + 2 == 4, discovering that 2 + 2 == 4 is true.

Realize that the condition (2 + 2 == 4 || whatever) must be true, no matter what the whatever condition happens to be.
Return true (without bothering to check if 0 == 1).

The condition \( 2 + 2 == 4 \) || whatever has to be true, because \( 2 + 2 == 4 \) is true. (Remember, the || operator wants either condition, on its left or right side or on both sides, to be true.)

So when Java finds the value on the left side of an || operator to be true, then Java declares the entire expression to be true.

Java's && and || operators use short circuit evaluation. Java's & and | operators also test for the "and" and "or" conditions, but these & and | operators don't do short circuit evaluation. In other words, when Java encounters the following code, Java checks to see if 0 == 1 is true and then, before giving its final answer, checks to see if 2 + 2 == 4 is true.

\[
\text{if (0 == 1 \&\& 2 + 2 == 4) }
\]

Here's a program to illustrate each operator's behavior:

```java
import static java.lang.System.out;

public class OperatorEvalDemo {
    public static void main(String args[]) {
        new OperatorEvalDemo();
    }

    OperatorEvalDemo() {
        if (0 == 1 \&\& 2 + 2 == 4) {
            out.println("(0 == 1 \&\& 2 + 2 == 4) is true");
        } else {
            out.println("(0 == 1 \&\& 2 + 2 == 4) is false");
        }
        out.println();

        if (2 + 2 == 4 || 0 == 1) {
            out.println("(2 + 2 == 4 || 0 == 1) is true");
        } else {
            out.println("(2 + 2 == 4 || 0 == 1) is false");
        }
        out.println();

        if (isFalse() \&\& isTrue()) {
            out.println("(isFalse() \&\& isTrue()) is true");
        } else {
            out.println("(isFalse() \&\& isTrue()) is false");
        }
    }
}
```
boolean isTrue() {
    out.println("Executing isTrue");
    return true;
}

boolean isFalse() {
    out.println("Executing isFalse");
    return false;
}
And here's the program's output:

\[(0 == 1 && 2 + 2 == 4)\] is false

\[(2 + 2 == 4 || 0 == 1)\] is true

Executing isFalse
\[(\text{isFalse}() \&\& \text{isTrue}())\] is false

Executing isFalse
Executing isTrue
\[(\text{isFalse}() \& \text{isTrue}())\] is false

Executing isTrue
\[(\text{isTrue}() || \text{isFalse}())\] is true

Executing isTrue
Executing isFalse
\[(\text{isTrue}() \mid\mid \text{isFalse}())\] is true

Notice, for example, what happens with the \&\& operator. Java displays \textbf{Executing isFalse}. But then Java doesn't display \textbf{Executing isTrue} because the \&\& operator does short circuit evaluation. On the other hand, Java displays both \textbf{Executing isFalse} and \textbf{Executing isTrue} for the \& operator, because the \& operator doesn't do short circuit evaluation.

You may wonder why anyone would use one kind of operator instead of another. Consider the following code:

```java
public class Oops {
    public static void main(String args[]) {
        Integer myInt;

        myInt = new Integer(42);
        if (myInt != null && myInt.intValue() == 42) {
            System.out.println("Comparing 42 to 42");
        }

        myInt = null;
        if (myInt != null & myInt.intValue() == 42) {
            System.out.println("Comparing null to 42");
        }
    }
}
```
Here's the code's output:

```
Comparing 42 to 42
Exception in thread "main" java.lang.NullPointerException
    at SideEffectDemo.main(SideEffectDemo.java:12)
```

This code checks twice to see if `myInt != null` and `myInt.intValue() == 42`. The first time around, the code uses short circuit evaluation. This is good because in this example, short circuit evaluation prevents Java from checking `myInt.intValue() == 42`.

But the second time around, the code doesn't use short circuit evaluation. No matter what happens when Java evaluates, `myInt != null`, the `&` operator marches on and evaluates `myInt.intValue() == 42`.

But here's the rub: If `myInt` has the value `null`, then the test is `myInt.intValue() == 42` destined to crash. This happens because you can't call a method (such as `intValue()`) on a null value. If you try, you get a `NullPointerException`. So in this example, the `&&` operator's short circuit evaluation saves you from crashing your program.

Occasionally you find situations in which you don't want short circuit evaluation. Usually these situations involve an evaluation's side effect. A side effect is something extra that happens during the evaluation of an expression. For example, in the `OperatorEvalDemo` program, displaying the line `Executing isTrue` is a side effect of evaluating the `isTrue()` expression.

Maybe, instead of displaying `Executing ...` lines, your methods check and make fine adjustments to a heart monitor and a lung monitor.

```
if (checkAdjustHeart() & checkAdjustLung()) {
    System.out.println("Both monitors are OK");
}
```

You may want to force Java to call both methods, even if the first method returns a `false` ("not OK") result. The `&&` operator's short circuit evaluation doesn't always call both methods. So in this scenario, you use the `&` operator.

The Hotel Example in Java For Dummies

Consider the following code (from `Java For Dummies`, 4th Edition):

```
import static java.lang.System.out;
import java.util.Scanner;
import java.io.File;
import java.io.IOException;
```
import java.io.PrintStream;

public class FindVacancy {

    public static void main(String args[])
        throws IOException {
            Scanner kbdScanner = new Scanner(System.in);
            Scanner diskScanner =
                new Scanner(new File("GuestList.txt"));
            int guests[] = new int[10];
            int roomNum;

            for (roomNum = 0; roomNum < 10; roomNum++) {
                guests[roomNum] = diskScanner.nextInt();
            }

            roomNum = 0;
            while (roomNum < 10 && guests[roomNum] != 0) {
                roomNum++;
            }

            if (roomNum == 10) {
                out.println("Sorry, no v cancy");
            } else {
                out.print("How many people for room ");
                out.print(roomNum);
                out.print("? ");
                guests[roomNum] = kbdScanner.nextInt();

                PrintStream listOut =
                    new PrintStream("GuestList.txt");

                for (roomNum = 0; roomNum < 10; roomNum++) {
                    listOut.print(guests[roomNum]);
                    listOut.println(" ");
                }
            }
        }

    The guests array is declared as follows:

    int guests[] = new int[10];
So there are elements named \texttt{guests[0]}, \texttt{guests[1]}, and so on up to (and including) \texttt{guests[9]}. There's no \texttt{guests[10]} element, so if Java tries to evaluate the expression

\[
guests[10] \neq 0
\]

then the program crashes with an \texttt{ArrayIndexOutOfBoundsException}. Now look at the \texttt{while} statement in the \texttt{FindVacancy} code:

\[
\text{while (roomNum < 10 && guests[roomNum] \neq 0) { }
\text{ roomNum++; }
}\]

What happens if the value of the \texttt{roomNum} variable is exactly 10? Then, because of the \&\& operator's short circuit evaluation, Java never evaluates the \texttt{guests[roomNum] \neq 0} expression. So the program doesn't crash.

But what if you reverse the tests in the \texttt{while} statement's condition?

\[
\text{while (guests[roomNum] \neq 0 && roomNum < 10) { }
\text{ roomNum++; }
}\]

Then the program can crash. Java evaluates boolean conditions from left to right. (This happens with both the short circuit \&\& and \| \| operators and with the non-short circuit \& and \| operators.) So before checking to make sure that \texttt{roomNum < 10}, Java evaluates the leftmost expression, \texttt{guests[roomNum] \neq 0}. Then Java tries to interpret \texttt{guests[10]} and crashes (because there's no \texttt{guests[10]} element).

The bottom line is, you must check \texttt{roomNum < 10} before you check \texttt{guests[roomNum] \neq 0}. To force Java to do the \texttt{roomNum < 10} check first, you put \texttt{roomNum < 10} on the left side of the \texttt{while} statement's condition. With \texttt{roomNum < 10} on the left side of the \&\& operator, short circuit evaluation prevents Java from accidentally evaluating \texttt{guests[roomNum] \neq 0} with \texttt{roomNum} equal to 10. Pretty slick, heh?