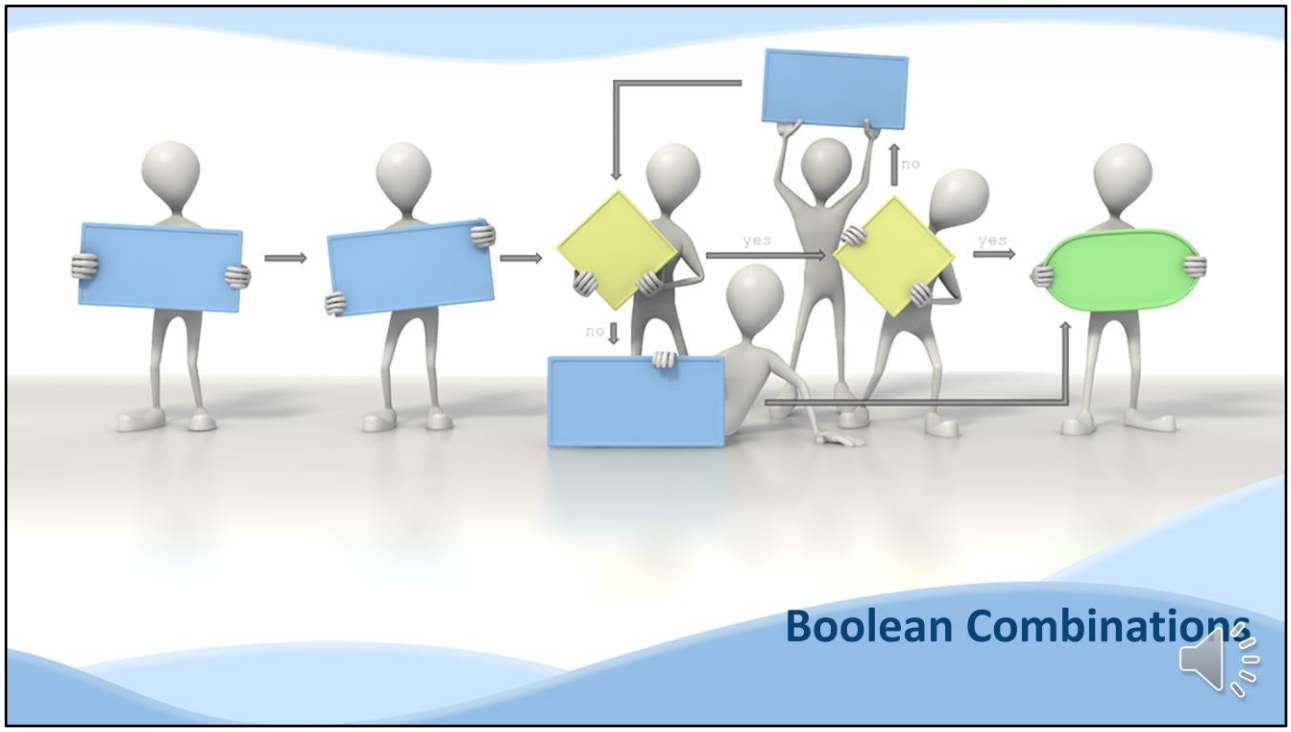
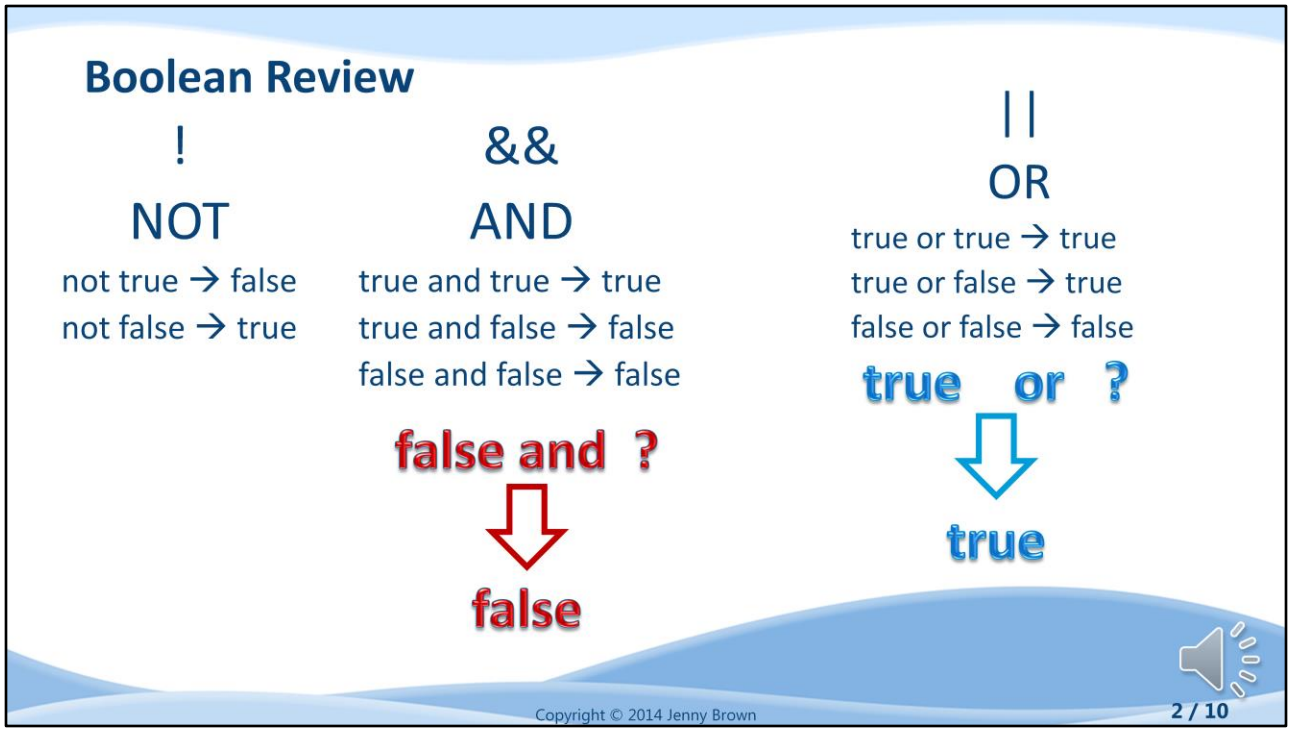
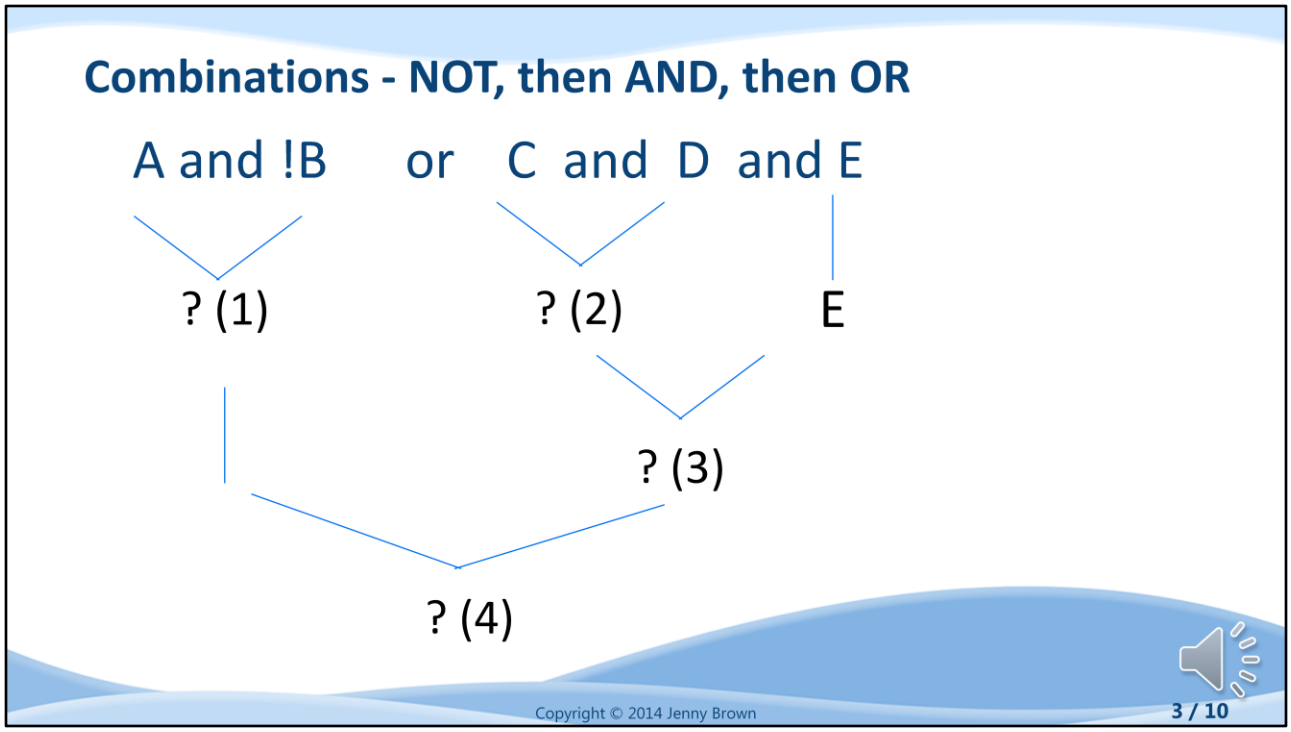


This video explores how to figure out logical combinations of several boolean values. For instance, if the sky is blue and the weather is warm, or the sky is orange and the weather is comfortable, then take my camera outside when I walk. How do we combine true and false values with lots of AND and OR operators?



This slide is a quick review of what you learned earlier about boolean logic. The NOT operator flips the value to its opposite; not true means false, while not false means true.

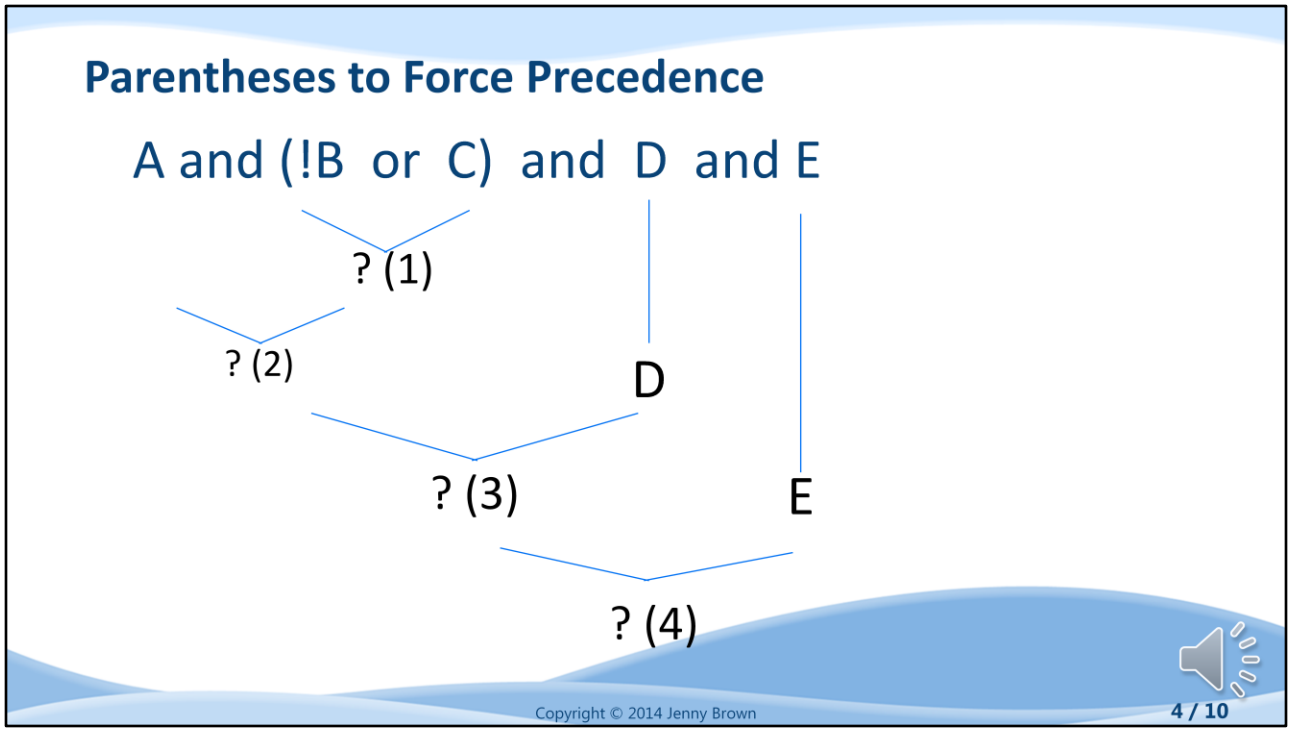
When you combine two values using AND, the result is true only when both of the values are true. When you combine two values using OR, the result is true when either of the values is true.



When we start to talk about more complex combinations, we need a way to know what order to work the problem in. This example uses boolean variables named A, B, C, D, E. When the computer is evaluating a complex combination like this, it first calculates the result of any NOT operators, then the AND operators, and finally the OR operators. Let's see how that works in this example.

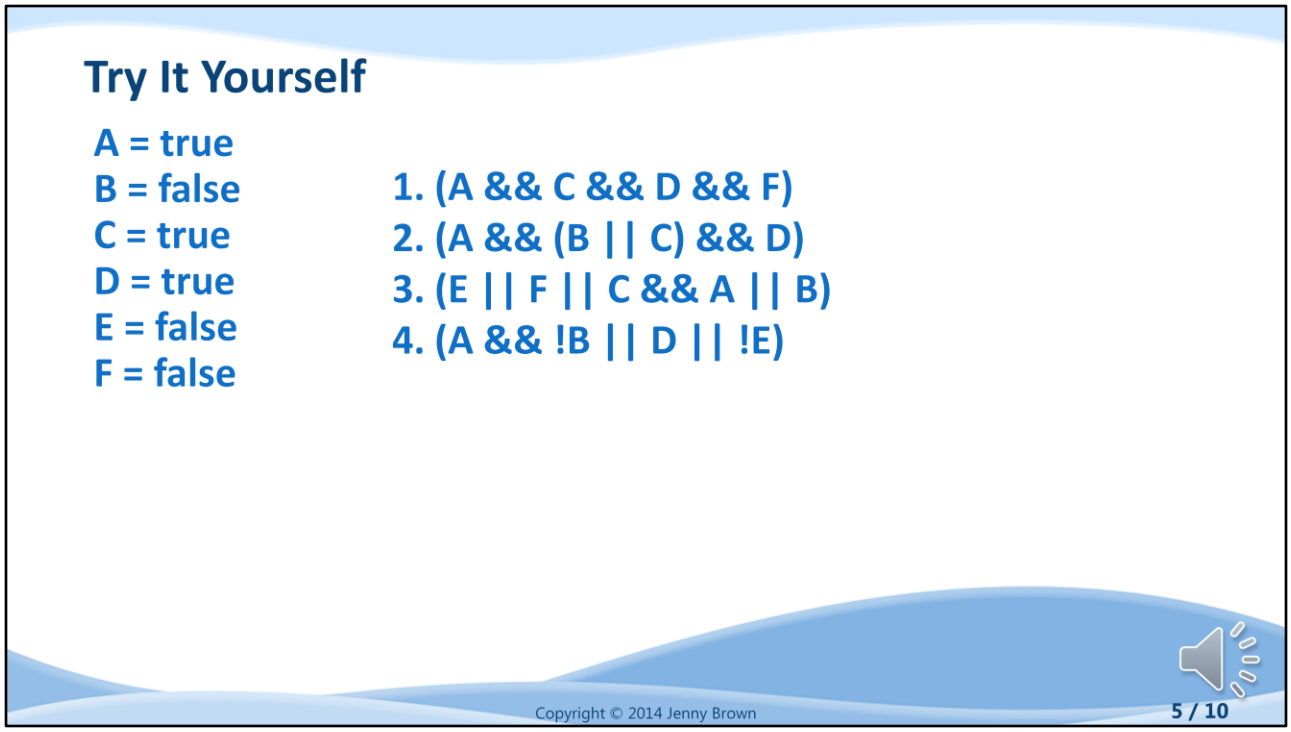
First, the computer calculates NOT-B. Then it looks ahead to the rest of the statement to check for any other NOTs. It doesn't find any. Then it starts from left to right to calculate the pairs bound by the AND operator. The first is A and NOT-B <click>, the second is C and D, <click> and the third is the result of that combined with E. <click> This completes the pass that covers the ANDs. Finally, it looks at what it has left, and calculates the OR <click> to finalize the result.

Notice how this lets you use the simple rules you saw on the previous slide, comparing only two values at a time. The computer never does a three-way compare. It only ever compares two values at a time. If it has to deal with more, it works in multiple steps, like shown here.



What do we do if the standard precendence, the standard order of operations, isn't what we need? We can force a particular order by using parentheses. In this example, NOT-B or C is calculated first <click> because the parentheses are given priority. Next, A and whatever that result was get combined <click> for the second step. In the third step, that result is combined as an AND with D <click>, and in the fourth step, combined as an AND with E <click> to finalize the answer.

You can always add parentheses if you need to force a particular order, or if you want your code to be more readable. You still need to learn the precendence rules, so you can read other people's code, which might only include parentheses when they are strictly necessary.



Here are a few practice problems for you to work out yourself. Grab a piece of paper and pause the video while you work. The answers are on the next slide.

Try It Yourself - Answers

A = true
B = false
C = true
D = true
E = false
F = false

1. (A && C && D && F) → false
2. (A && (B || C) && D) → true
3. (E || F || C && A || B) → true
4. (A && !B || D || !E) → true

How did you do? If you missed any, review the rules about what order to do calculations in, and recheck your work. Draw a diagram like the earlier slides, if you need to. Be patient; this stuff takes some practice!

Comparisons

What does this print?

```
int c = 5;
int d = 3;
String word = "bird";
if (c > 4 && d<=7 || word.equals("bird") && true == !false)
{
  out.println("Running");
} else {
  out.println("Skipped");
}
```

When you are coding, you often won't have boolean variables to compare. Instead, you'll have boolean true/false results from comparing other kinds of variables. Here's a more realistic example. Try to work out what this code prints. The answer is in the next slide.

Comparisons

What does this print?

```
int c = 5;
int d = 3;
String word = "bird";
if (c > 4 && d<=7 || word.equals("bird") && true == !false)
{
  out.println("Running");
} else {
  out.println("Skipped");
}
```

How did you do? Recheck your work carefully if you missed anything.

Boolean Algebra

Boolean logic can be regrouped just like in math class.

1. (A and B) and C → A and (B and C)
2. (A or B) or C → A or (B or C)
3. !(!B) → B
4. A and (B or C) → (A and B) or (A and C)
5. A or (B and C) → (A or B) and (A or C)
6. !(B or C) → !B and !C
7. !(B and C) → !B or !C

Boolean logic also includes some ways of re-writing a boolean expression. This can be handy sometimes for making your code more tidy and easier to read, or even for figuring out what someone else's code actually means. Line 1 and line 2 show that the order doesn't matter when you're just doing ANDs alone, or when you're just doing ORs alone. Line 3 shows that when you use NOT to flip the value twice, it's back to where it started.

Line 4 demonstrates how the AND can be distributed across the B, C inside the parentheses, without changing the meaning of the statement. Line 5 demonstrates how to distribute the OR across an AND in parentheses. Line 6 shows how the NOT distributes through the parentheses. Notice that the OR changed to an AND. This is a really important nuance. Line 7 shows the same kind of behavior, this time switching the AND to an OR.

Memorizing this set of rules would be time well spent. It takes some practice to get them all firmly into your memory. You will use them throughout your years of programming, and the effort of memorizing them will deepen your understanding of complex boolean combinations. By memorizing it now, you'll make fewer mistakes later, when you use it in code.

Journal, Reflect, Remember

Was it harder to do numerical comparisons and then think about their boolean results? Why or why not?

How did you feel about boolean algebra? Tedium? Uncertainty? Curiosity?

Was anything unclear?

Journal time.

Was it harder to do numerical comparisons and then think about their boolean results? Why or why not?

How did you feel about boolean algebra? Tedium? Uncertainty? Curiosity?

Was anything unclear?