Object-Oriented Programming Fundamentals

Lecture/Workshop (Week 3)

Boolean logic

Two values true and false

Example

3 > 4 is false
3 <= 4 is true

Boolean operator ‘and’

In Java the operator is: `&&`

Truth table for &&

e1	e2	e1 && e2
true	true	**true**
true	false	**false**
false	true	**false**
false	false	**false**

From the truth table, we can see that in an expression with all **`&&`**, if just one condition is **`false`**, the whole condition is **`false`**, regardless of how many other expressions are **`true`**.

Boolean operator ‘or’

In Java the operator is: `||`

Truth table for ||

e1	e2	e1 || e2
true	true	**true**
true	false	**true**
false	true	**true**
false	false	**false**

From the truth table, we can see that in an expression with all **`||`**, if just one condition is **`true`**, the whole condition is **`true`**, regardless of how many other expressions are **`false`**.

Boolean operator ‘not’

In Java the operator is: `!`

Truth table for !

e1	! e1
true	**false**
false	**true**

Evaluate the condition first, then as the last step reverse the answer.

Boolean operator 'exclusive or' (XOR) (optional reading – for interest)

In Java the operator is: ^

Sometimes we wish to find out if, or assert that, exactly one of two things is true, that is one is false and the other is true

For example

The train is on track 1 or track 2 (but cannot be on both)

Note we can write an equivalent expression for exclusive or using && and ||

```
condition1 ^ condition2 →

        condition1 && !condition2 || !condition1 && condition2
```

Task 1

What is the value of the following expressions if **`count`** is 0 and **`limit`** is 10? (The expressions are boolean expressions that evaluate to **`true`** or **`false`**. To determine the value of the expression, evaluate each operand, then apply the operator as in the truth tables.)

```
(count == 0) && (limit < 20)
```

```
(limit > 20) || (count < -5)
```

```
!(count == 12)
```

```
(count <= 0) || (limit <= limit)
```

```
(count < 0) && (limit <= limit)
```

Precedence rules (order of operations)

What is the value of this expression?

```
9 > 6 || 0 == 0 && 7 == 6
```

For arithmetic expressions we know that '*' has precedence over '+'

So 3 + 4 * 6
is equivalent to 3 + (4 * 6)

Boolean operators must also be given a precedence

Java operator precedence from highest to lowest

```
+   -   ++   --   !      (unary operators)
*   /   %
+   -                    (binary +, - operators)
<   >   <=   >=
==   !=
&&
||
```

Brackets below illustrate the order of operations applied to the expression

```
  9 > 6 || 0 == 0 && 7 == 6
(9 > 6) || 0 == 0 && 7 == 6
(9 > 6) || (0 == 0) && (7 == 6)
(9 > 6) || ((0 == 0) && (7 == 6))
```

Task 2

What is the value of the following expressions where `count` is 0 and `limit` is 10?

```
count == 0 && limit < 20
```

```
count > 0 && limit > 20 || limit > 0
```

```
count > 0 || limit > 20 && limit > 0
```

```
3 + 4 > 4 && count != 0
```

Short-circuit evaluation

Java uses short-circuit evaluation

- If the first part of an `||` is **`true`**, the second part of the `||` is not evaluated (as the whole expression *must* be **`true`**)
- If the first part of an **`&&`** is **`false`**, the second part of the **`&&`** is not evaluated (as the whole expression *must* be **`false`**)

Example

```
int kids = 0;
if ( (kids != 0) && ((pieces / kids) >= 2) )
{
    ...
}
```

In Java, this operand will *not* be evaluated as the first operand is **`false`** and the logical operator is **`&&`** – so we avoid a run-time error of division by zero!

Task 3

What is the value of the following expressions where **`count`** is 0 and **`limit`** is 10?

```
(count == 1) && (x < y)
```

```
(limit < 20) || ((limit/count) > 7)
```

```
(count > 20) && ((limit/count) > 7)
```

```
(limit < 20) && ((limit/count) > 7)
```

Equivalent expressions

Some boolean expressions can be expressed in various equivalent forms

Choose the one easiest to understand (if possible)

For example, both these expressions area the same;

```
int x = 3;
System.out.println(x <= 5 && x > 0);
System.out.println(!(x <= 0) && !(x > 6));
```

Relational operators (example)

How can we decide to do something if **`time`** is not greater than **`limit`**?

Assume **`time`** is 20 and **`limit`** is 30

Attempt 1

```
if (!time > limit)
{
   System.out.println("time for another game");
}
```

```
!time - compile-time error
operator ! cannot be applied to int
      if (!time > limit)
          ^
```

Problems? *Look at the order of operations (precedence)* **`!time`** *is a boolean operator acting on an integer type – this will not compile*

Attempt 2 – This attempt is correct but the logic is not easy to understand

```
if (!(time > limit))
{
   System.out.println("time for another game");
}
```

Attempt 3 – Easy to understand logic – avoid using ! where possible

```
if (time <= limit)
{
   System.out.println("time for another game");
}
```

Rules for distributing ! over relational operators

! (a >= b) ⇔ a < b
! (a > b) ⇔ a <= b
! (a <= b) ⇔ a > b
! (a < b) ⇔ a >= b

! (a == b) ⇔ a != b
! (a != b) ⇔ a == b

Task 4
Rewrite without `!`

```
!(numberOfGames > 5)
```

```
!(balance + interest <= 2000)
```

```
!(userInput != 'q')
```

De Morgan's law

! (a && b) ⇔ !a || !b
! (a || b) ⇔ !a && !b

Task 5
Distribute the ! over the operators

```
!(apples || oranges)
```

```
!(input == 'y' && tries < 5)
```

```
!(input != 'n' && tries >= 6)
```

Double negation
Two negations cancel one another

```
!!a  ⇔  a
```

Task 6
Simplify

```
!(!(tries > 5))
```

```
!!(bananas != 6)
```

Boolean output with System.out

Will output the word **`true`** or **`false`**

```
boolean b;
b = x > y;
System.out.println(b);
```

Boolean input with the Scanner class

Expects true, false, (will accept uppercase and mixed upper and lowercase)

```
b = keyboard.nextBoolean();
```

Challenge 1 (Optional)

```
if (                                                    )
{
   System.out.println("Fred enjoyed the movies");
}
else
{
   System.out.println("Fred didn't enjoy the movies");
}
```

What should the condition be if:

- Fred always enjoys the movie if he goes on a Tuesday
- If Fred goes with Nicola, then he enjoys the movie
- If Fred goes with anyone but Nicola, he only enjoys the movie if he sees a science fiction or an action movie

Assume the integer variable **`day`** stores the day on which he went (1 = Monday, 2 = Tuesday, etc.)

Assume the String variable **`companion`** stores the name of the person with whom he went

Assume the char variable **`movieType`** stores the type of movie he saw ('a' = action, 's' = science fiction, 'c' = comedy, 'r' = romance)

What should the condition be if the problem is changed to the following?

- Fred always enjoys the movie if he goes on a Tuesday
- If Fred goes with Nicola, then he only enjoys a comedy or romance
- If Fred goes with anyone but Nicola, he only enjoys the movie if he sees a science fiction or an action movie

Challenge 2 (Optional) – Summary of Bitwise operators (for interest)

Used for low-level bitwise manipulation of data (which are represented as integers)

Examples are

&	bitwise AND
|	bitwise OR
^	bitwise XOR
~	bitwise complement
<<	left shift
>>	arithmetic right shift
>>>	bitwise left shift

e.g. 0xF0F0 & 0x0F0F gives 0x0000 (i.e. 0)

Boolean operands with &, | and ^

When & or | are used with boolean operands, it does a boolean test but no short circuit evaluation occurs, i.e. both operands are fully evaluated (complete evaluation).

When ^ is used with boolean operands, we have boolean XOR, true if exactly one of the two operands is true.

Operator	Operand type	Result
&&	boolean	Boolean 'and' with short circuit evaluation
&	boolean	Boolean 'and' with complete evaluation
&	integer types	Bitwise 'and'
||	boolean	Boolean 'or' with short circuit evaluation
|	boolean	Boolean 'or' with complete evaluation
|	integer types	Bitwise 'or'
^	boolean	Boolean XOR
^	Integer types	Bitwise XOR

Some optional questions on bitwise operators are in this week's lab ☺!

Wanting help? Meet with a demonstrator and fellow students in one of the voluntary help sessions to exchange ideas and work through problems together.

to be advised