Logical Operators and switch

Stuart Gibson
sg@sys.uea.ac.uk

Examples of Simple Logical Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &lt; b</td>
<td>true</td>
</tr>
<tr>
<td>a == c</td>
<td>true</td>
</tr>
<tr>
<td>c &gt; b</td>
<td>false</td>
</tr>
<tr>
<td>choice == no</td>
<td>false</td>
</tr>
<tr>
<td>c != a</td>
<td>false (Incorrect in Lecture 4 @)</td>
</tr>
</tbody>
</table>

• In the last lecture we saw how these logical expressions are evaluated at run time as either true or false.

Evaluation of logical expressions

```java
int a=10, b=20, c=5;
if( a < b )
{
    if ( a > c )
    {
        System.out.println("Second if");
    }
    System.out.println("End of first if");
}
else
{
    System.out.println("In else");
}
System.out.println("After");
```

Relational and Equivalence Operators

<table>
<thead>
<tr>
<th>Relational Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Equivalence Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>equal to</td>
</tr>
<tr>
<td>!=</td>
<td>not equal to</td>
</tr>
</tbody>
</table>

• A variable may be tested against a constant or another variable.
Logical Operators

- The logical operators **AND**, **OR** and **NOT** can be used in Java to form more complex logical expressions by combining simple logical expressions.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>AND</td>
<td>Two Arguments</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>NOT</td>
<td>One Argument</td>
</tr>
</tbody>
</table>

The AND Operator (**&&**)

- Example:
  ```java
  if ( x >= 0 && x <= 10)
      System.out.print("x in range 0 to 10");
  System.out.print("x in range 0 to 10");
  ```

- An AND expression evaluates as **true** only if BOTH arguments are **true**.

- Truth Table:

<table>
<thead>
<tr>
<th>Argument 1</th>
<th>Argument 2</th>
<th>Argument 1 &amp;&amp; Argument 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>
The AND Operator (&&) - Examples

```java
int x = 10, y = 11, z = -3;
if (x > z && x > y)
  \(\rightarrow (10 > -3 \&\& 10 > 11)\)
  \(\rightarrow (true \&\& false)\)
  \(\rightarrow false\)
if (z < x && y > x)
  \(\rightarrow (-3 < 10 \&\& 11 > 10)\)
  \(\rightarrow (true \&\& true)\)
  \(\rightarrow true\)
if (z > x && x > y)
  \(\rightarrow (-3 > 10 \&\& 10 > 11)\)
  \(\rightarrow (false \&\& false)\)
  \(\rightarrow false\)
```

The OR Operator (||) - Examples

```java
int x = 10, y = 11, z = -3;
if (x > z || x > y)
  \(\rightarrow (10 > -3 || 10 > 11)\)
  \(\rightarrow (true || false)\)
  \(\rightarrow true\)
if (z < x || y > x)
  \(\rightarrow (-3 < 10 || 11 > 10)\)
  \(\rightarrow (true || true)\)
  \(\rightarrow true\)
if (z > x || x > y)
  \(\rightarrow (-3 > 10 || 10 > 11)\)
  \(\rightarrow (false || false)\)
  \(\rightarrow false\)
```

Evaluation of && and || Operator

- The computer is quite clever, so when evaluating && and || operators in some cases it only evaluates the first argument. For example:
  ```java
  int a = 0;
  if (a > 0 && b / a > 1)
  ```
- In an AND (&&) statement, if the first argument is false, then whatever the second argument is, doesn't matter! This is because according to the truth table, any combination starting with false evaluates to false.
- In an OR (||) statement, if the first argument evaluates to true, then whatever the second argument is, doesn't matter! This is because and combination starting with true, evaluates to true.

<table>
<thead>
<tr>
<th>Argument 1</th>
<th>Argument 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>
The NOT Operator ( ! )

- Example:
  ```java
  if ( ! ( x == y ) )
  {
      System.out.print("x is not equal to y");
      System.out.print("easier to use x!=y in");
      System.out.print(" this case.");
  }
  ```

- A NOT expression is true when its argument is false.

- Truth Table:
<table>
<thead>
<tr>
<th>Argument</th>
<th>! Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>

Compound Logical Expressions

- We can build more complex expressions containing more than one logical operator.
  ```java
  int x = 1, y = 2, z =3;
  x == 1 || y == 2 && z == 4
  ```

- Unless parentheses (brackets) are used AND is always evaluated before OR so:
  ```java
  x == 1 || y == 2 && z == 4
  ```
  ```java
  true || true && false
  ```
  ```java
  true && false
  ```
  ```java
  false
  ```

- However the expression to evaluate can be even more complicated such as:
  ```java
  if ( !( x > y && y < z ) && z > 0 || (y>=12) )
  ```
### Compound Logical Expressions

```java
if ( !( x > y && y < z) && z > 0 || (y>=12))
```

- **Q:** How does the computer know what to evaluate first?  
  **A:** Precedence!

- Brackets have the highest precedence. Always solve the deepest nested set first (L:N where N is the level of nesting):

  ```java
  if ((x >5 && (y < z || y > 3)) && (a > 3))
  ```

  - So, solve level 3 (deepest nested) first, then level2, note ties on the same level are solved left to right and finally level 1.

### Precedence Table

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>Logical not, unary plus and minus</td>
</tr>
<tr>
<td>+, -</td>
<td>Addition and subtraction</td>
</tr>
<tr>
<td>* / %</td>
<td>Multiplication, division and mod</td>
</tr>
<tr>
<td>&lt; &lt;= &gt;</td>
<td>Relational operators</td>
</tr>
<tr>
<td>== !=</td>
<td>Equivalence operators</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>Assignment</td>
</tr>
</tbody>
</table>

### More Logical Expressions

- Logical expression can involve arithmetic operators:

  ```java
  if ((x = y + 5) > 10 && (z = z + y) == 4))
      System.out.print("Some condition met");
  ```

- But it is much better to write:

  ```java
  x = y + 5;
  z = z + y;
  if (x > 10 && z == 4)
      System.out.print("Some condition met");
  ```

### Character Comparisons

- You can compare characters as well:

  - `A' < 'B'` true
  - `A' < 'a'` true
  - `?' > 'g'` false

- **Q:** Where does those values come from?  
  **A:** The computer converts the characters to numbers and then compares those numbers.

- The numbers come from the ASCII / UNICODE values:  
  `A = 65, B = 66, a = 97, ? = 63, @ = 64.`
Control Structures

• In the last lecture we saw that there were three main types of Control Structures:
  • Sequence
  • Selection
  • Repetition

• Selection

So far we have seen the following Java control sequences:

One Alternative: \textit{if} statement

Two Alternatives: \textit{if else} statement

More than two Alternatives: nested \textit{if else} statements

Selection: \textit{switch}

• \textit{switch} statements can be used to chose between one of several alternatives.

• The alternatives are dependant on the value of a \textbf{selector}

• The selector must be an expression that evaluates to an integer (\texttt{int}), character (\texttt{char}), or boolean.

• Note that with boolean selector you can only have two choices, true or false, like a two alternative, \textit{if else} statement.

\begin{verbatim}
switch (selector)
{
  case label_1:
    statements_1;
    break;
  case label_2:
    statements_2;
    break;
  ... 
  case label_n:
    statements_n;
    break;
  default:
    statements_d;
}
\end{verbatim}
### switch statement

```java
char staffType;
...
switch (staffType)
{
    case 'A':
        System.out.print("Academic Staff");
        break;
    case 'U':
        System.out.print("Undergraduate Student");
        break;
    case 'P':
        System.out.print("Postgraduate Student");
        break;
    case 'T':
        System.out.print("Technical Staff");
        break;
    default :
        System.out.print("Unknown Staff Type");
}
```

### Execution of a switch statement

- The selector expression is evaluated and compared to each case label in turn, for example:
  ```java
  staffType = 'P';
  case 'A': \( \rightarrow \) staffType == 'A' \( \rightarrow \) 'P' == 'A' \( \rightarrow \) false
  case 'U': \( \rightarrow \) staffType == 'U' \( \rightarrow \) 'P' == 'U' \( \rightarrow \) false
  case 'P': \( \rightarrow \) staffType == 'P' \( \rightarrow \) 'P' == 'P' \( \rightarrow \) true
  ```
- If the selector is equal to the condition of one of the case labels, then the statements associated to this case are executed, so in our example the output would be:

```
Postgraduate Student
```

### switch – the use of break

- All statements after the matched case label are executed until a break statement is reached.

- The purpose of the break statement is to break out of the current control structure, in this case the switch statement.

- In our example where staffType = 'P'; the program executes the one line of code associated with the case 'P': and then breaks out of the switch. Program execution continues at the next statement after the last curly brace of the switch statement.

### switch – with break

```
char staffType;
staffType = 'P';
switch (staffType)
{
    case 'A':
        System.out.print("Academic Staff");
        break;
    case 'U':
        System.out.print("Undergraduate Student");
        break;
    case 'P':
        System.out.print("Postgraduate Student");
        break;
    case 'T':
        System.out.print("Technical Staff");
        break;
    default :
        System.out.print("Unknown Staff Type");
}
```

Output

```
Postgraduate Student
```

Flow of execution
The code snippet demonstrates the use of a `switch` statement with different case labels and the effect of missing `break` statements.

### switch - missing break

```java
char staffType;
staffType = 'P';
switch (staffType)
{
    case 'A':
        System.out.print("Academic Staff");
        break;
    case 'U':
        System.out.print("Undergraduate Student");
        break;
    case 'P':
        System.out.print("Postgraduate Student");
        break;
    case 'T':
        System.out.print("Technical Staff");
        break;
    default :
        System.out.print("Unknown Staff Type");
}
System.out.println("After Switch");
```

### switch - default case

In our example, what would happen if `staffType` was 'Z'?

```java
staffType = 'Z';
```

- `case 'A':` → `staffType == 'A'` → `false`
- `case 'U':` → `staffType == 'U'` → `false`
- `case 'P':` → `staffType == 'P'` → `true`
- `case 'T':` → `staffType == 'P'` → `false`
- `default :` → `true`

If none of the case labels are matched, the statements associated with the `default` case are executed.

A `default` case is optional in a `switch` statement.

### switch - definition errors

- Each case label is a single, constant value and each label must be different.

```java
case 'U':
    System.out.print("Undergraduate Student");
    break;
```

- Gives the following error:

```
HelloWorld.java:15: duplicate case label
  case 'U':
       ^
1 error
```

### switch with integers (int)

```java
int numWheels;
... ...
switch (numWheels)
{
    case 1:
        System.out.print("Unicycle");
        break;
    case 2:
        System.out.print("Bike");
        break;
    case 3:
        System.out.print("Trike");
        break;
    case 4:
        System.out.print("Stabilisers");
        break;
    default :
        System.out.print("Weird Bike!");
}
```
**Multiple cases**

- You can have multiple cases associated with a single set of statements:
  ```java
  int employeeCode = 3;
  switch (employeeCode) {
    case 0: case 1:
      salary = salaryLevel_1;
      break;
    case 2: case 3: case 4:
      salary = salaryLevel_2;
      break;
    case 5:
      salary = managerLevel;
      break;
    default :
      System.out.print("Unknown Employee Code");
  }
  ```

- However each case has to be specified individually in a list as you can not specify a range of values.

**Multiple cases – NO range!**

- For example trying to specify a range would give you an error:
  ```java
  int employeeCode = 3;
  switch (employeeCode) {
    case employeeCode <=1:
      ...  
  }
  ```

HelloWorld.java:9: incompatible types
found : boolean
required: int  
    case employeeCode <=1:  
  ^

1 error

**switch – case sensitive**

- Remember Java is case sensitive!

- If your `switch` selector is a character, you need to deal with both upper and lower case letters, for example:
  ```java
  case 'U': case 'u':
    System.out.print("Undergraduate Student");
    break;
  ```

- A more advanced solution would be to convert the selector to uppercase or lowercase before entering the `switch` statement, more later...