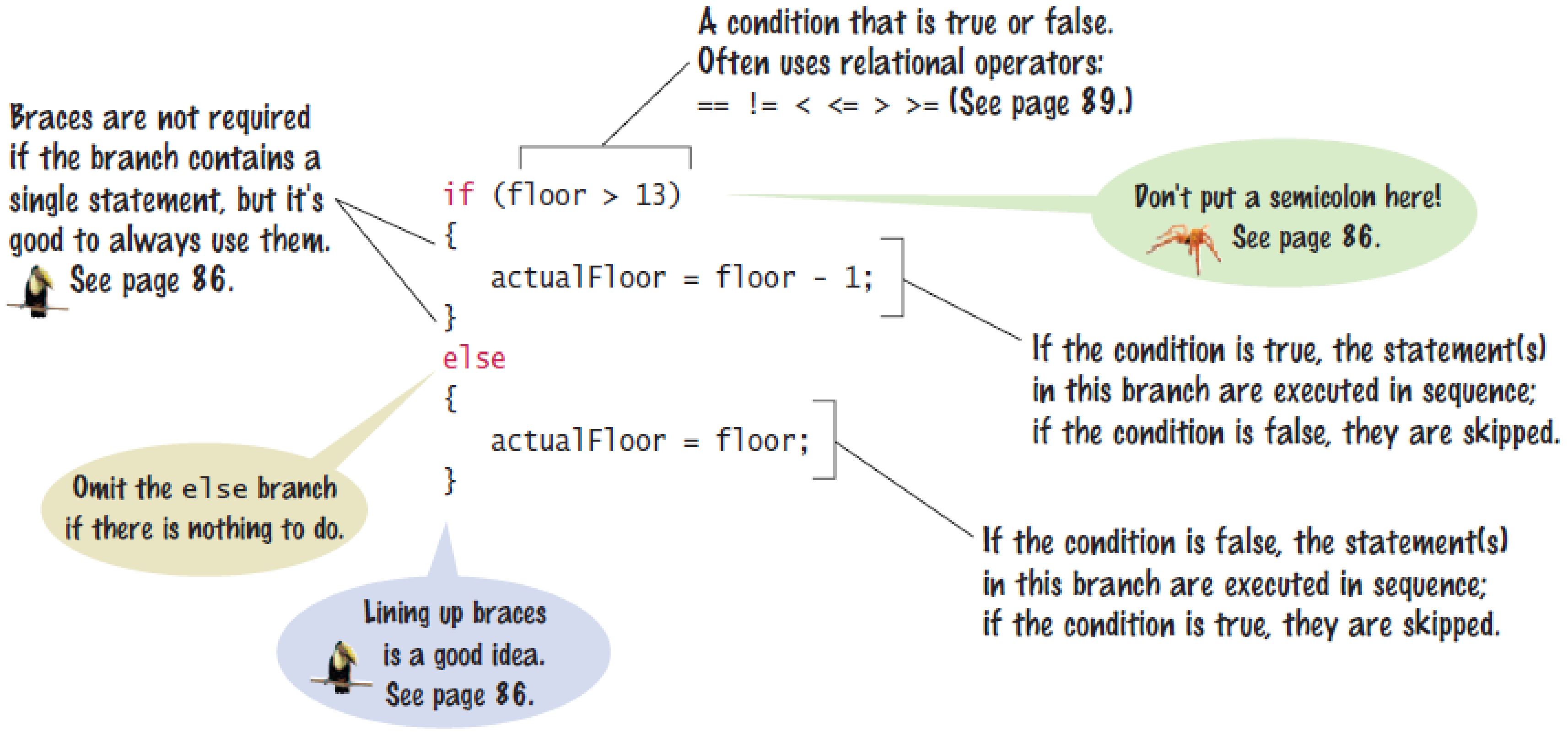
CHAPTER 3

DECISIONS

Slides by Donald W. Smith
TechNeTrain.com

Final Draft
Oct 15, 2011

Syntax 3.1: The `if` statement



A condition that is true or false. Often uses relational operators: == != < <= > >= (See page 89.)

Braces are not required if the branch contains a single statement, but it's good to always use them. See page 86.

Don't put a semicolon here! See page 86.

```
if (floor > 13)
{
   actualFloor = floor - 1;
}
else
{
   actualFloor = floor;
}
```

If the condition is true, the statement(s) in this branch are executed in sequence; if the condition is false, they are skipped.

If the condition is false, the statement(s) in this branch are executed in sequence; if the condition is true, they are skipped.

Omit the else branch if there is nothing to do.

Lining up braces is a good idea. See page 86.

Tips On Using Braces

- Style: line up all pairs of braces vertically
  - Lined up

```
if (floor > 13)
{
   floor--;
}
```

Not aligned (saves lines)

```
if (floor > 13) {
   floor--;
}
```

- Always use braces (even when not required)
  - Although single statement clauses do not require them

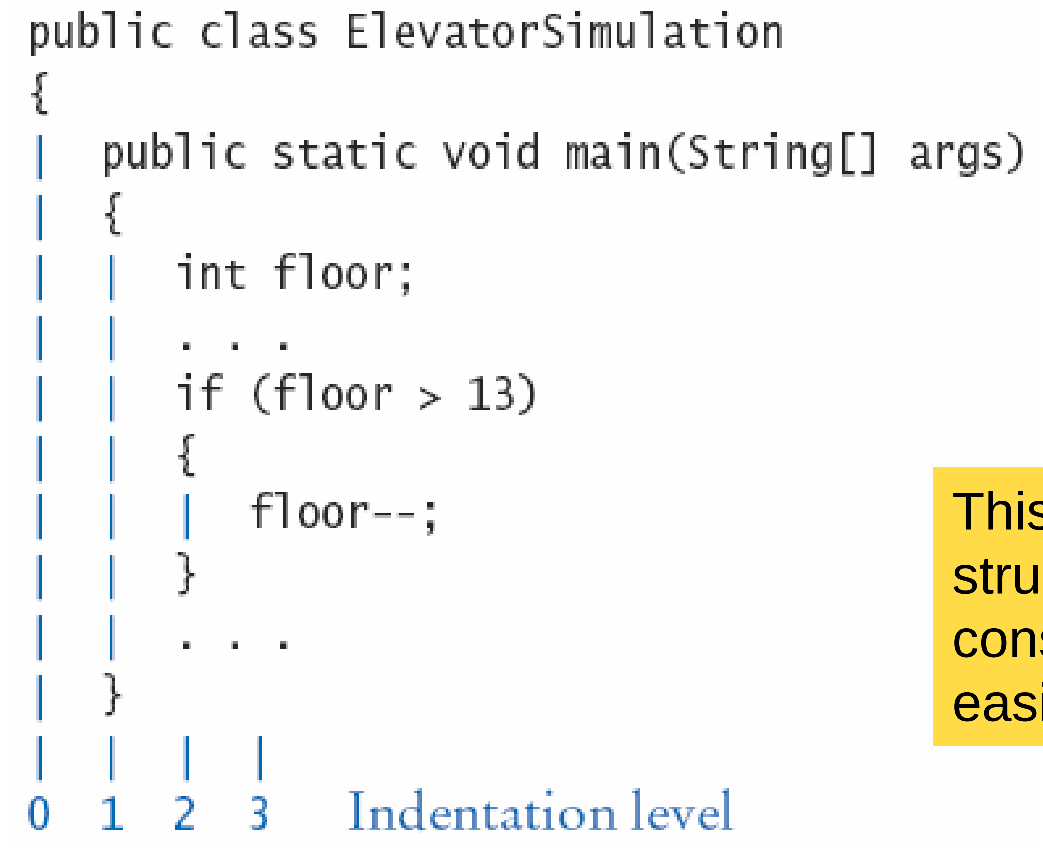
```
if (floor > 13)
{
   floor--;
}
```

```
if (floor > 13)
   floor--;
```

Most programmer's editors and IDEs have a tool to align matching braces.

Style tips on indenting blocks

- Use Tab to indent a consistent number of spaces



```
public class ElevatorSimulation
{
|   public static void main(String[] args)
|   {
|   |   int floor;
|   |   . . .
|   |   if (floor > 13)
|   |   {
|   |   |   floor--;
|   |   }
|   |   . . .
|   }
|   |   |   |
0   1   2   3    Indentation level
```

This is referred to as 'block-structured' code. Indenting consistently makes code much easier for humans to follow.

Common Error: A semicolon after an if statement

It is easy to forget and add a semicolon after an `if` statement.

- The true path is now the space just before the semicolon

```
if (floor > 13) ;
{
   floor--;
}
```

- The 'body' (between the curly braces) will always be executed in this case

The Conditional Operator

- A ‘shortcut’ you may find in existing code
  - It is not used in this book

$$\texttt{actualFloor = }\overbrace{\texttt{floor > 13}}^{\text{Condition}}\texttt{ ? }\overbrace{\texttt{floor - 1}}^{\text{True branch}}\texttt{ : }\overbrace{\texttt{floor}}^{\text{False branch}}\texttt{;}$$

  - Includes all parts of an if-else clause, but uses:
    - ? To begin the true branch
    - : To end the true branch and start the false branch

Comparing Numbers and Strings

- Every `if` statement has a condition
  - Usually compares two values with an operator

```
if (floor > 13)
   ..
if (floor >= 13)
    ..
if (floor < 13)
   ..
if (floor <= 13)
    ..
if (floor == 13)
   ..
```

Beware!

Table 1 Relational Operators

Java	Math Notation	Description
>	$>$	Greater than
>=	$\geq$	Greater than or equal
<	$<$	Less than
<=	$\leq$	Less than or equal
==	$=$	Equal
!=	$\neq$	Not equal

Operator Precedence

- The comparison operators have lower precedence than arithmetic operators
  - Calculations are done before the comparison
  - Normally your calculations are on the ‘right side’ of the comparison or assignment operator

$$\texttt{actualFloor = }\overbrace{\texttt{floor + 1}}^{\text{Calculations}}\texttt{;}$$

```
if (floor > height + 1)
```

Relational Operator Use (1)

Table 2 Relational Operator Examples

Expression	Value	Commment
`3 <= 4`	`true`	3 is less than 4; <= tests for "less than or equal".
🚫 `3 =< 4`	**Error**	The "less than or equal" operator is <=, not =<. The "less than" symbol comes first.
`3 > 4`	`false`	> is the opposite of <=.
`4 < 4`	`false`	The left-hand side must be strictly smaller than the right-hand side.
`4 <= 4`	`true`	Both sides are equal; <= tests for "less than or equal".
`3 == 5 - 2`	`true`	== tests for equality.
`3 != 5 - 1`	`true`	!= tests for inequality. It is true that 3 is not 5 – 1.

Relational Operator Use (2)

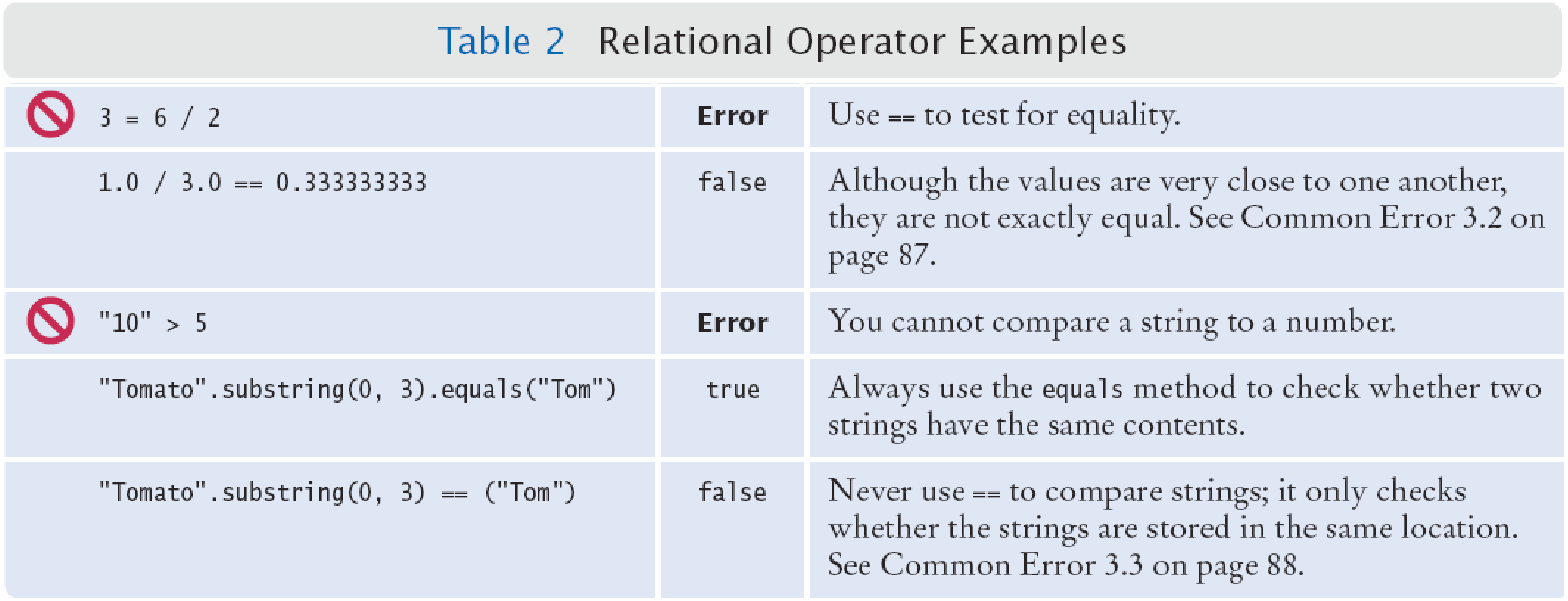


Table 2 Relational Operator Examples

🚫	`3 = 6 / 2`	**Error**	Use == to test for equality.
	`1.0 / 3.0 == 0.333333333`	`false`	Although the values are very close to one another, they are not exactly equal. See Common Error 3.2 on page 87.
🚫	`"10" > 5`	**Error**	You cannot compare a string to a number.
	`"Tomato".substring(0, 3).equals("Tom")`	`true`	Always use the `equals` method to check whether two strings have the same contents.
	`"Tomato".substring(0, 3) == ("Tom")`	`false`	Never use == to compare strings; it only checks whether the strings are stored in the same location. See Common Error 3.3 on page 88.

Comparing Strings

Comparing Strings

- Strings are a bit 'special' in Java (More on Strings later)
- Do not use the == operator with Strings!!!
  - The following compares the memory locations of two strings, and not their contents

```
if (string1 == string2) ...
```

- Instead use the String's `equals` method:

```
if (string1.equals(string2)) ...
```

Common Error

- Using == to compare Strings
  - == compares the **memory** ***locations*** of the Strings
- Java creates a new String every time a new word inside double-quotes is used
  - Exception: If there is one that matches it exactly, Java re-uses it

```
String name = "Robert";
String nickname = name.substring(0, 3);
. . .
if (nickname == "Rob") // Test is false
```

Comparing Strings: Lexicographical Order

- To compare Strings in 'dictionary' order
  - When compared using `compareTo`, string1 comes:
    - Before `string2` if `string1.compareTo(string2) < 0`
    - After `string2` if `string1.compareTo(string2) > 0`
    - Equal to `string2` if `string1.compareTo(string2) == 0`
  - Notes
    - All UPPERCASE letters come before lowercase
    - 'space' comes before all other printable characters
    - Digits (0-9) come before all letters
    - See Appendix A for the Basic Latin Unicode (ASCII) table

Comparing Basic Types

Comparing Characters char

- char c1 = ‘a’;
- char c2 = ‘2’;

- if (c1 == c2) …..

Common Error: Comparing Floats/Doubles

- Comparison of Floating-Point Numbers
  - Floating-point numbers have limited precision

```
double r = Math.sqrt(2.0);
if (r * r == 2.0)
{
   System.out.println("Math.sqrt(2.0) squared is 2.0");
}
else
{
   System.out.println("Math.sqrt(2.0) squared is not 2.0
    but " + r * r);
}
```

```
Output:
Math.sqrt(2.0) squared is not 2.0 but 2.00000000000000044
```

The use of EPSILON

- Use a very small value to compare the difference if floating-point values are ‘*close enough*’
  - The magnitude of their difference should be less than some threshold
  - Mathematically, we would write that x and y are close enough if:

$$|x - y| < \varepsilon$$

```
final double EPSILON = 1E-14;
double r = Math.sqrt(2.0);
if (Math.abs(r * r - 2.0) < EPSILON)
{
   System.out.println("Math.sqrt(2.0) squared is approx. 
    2.0");
}
```

3.3 Multiple Alternatives

- What if you have more than two branches?
- Count the branches for the following earthquake effect example:
  - 8 (or greater)
  - 7 to 7.99
  - 6 to 6.99
  - 4.5 to 5.99
  - Less than 4.5

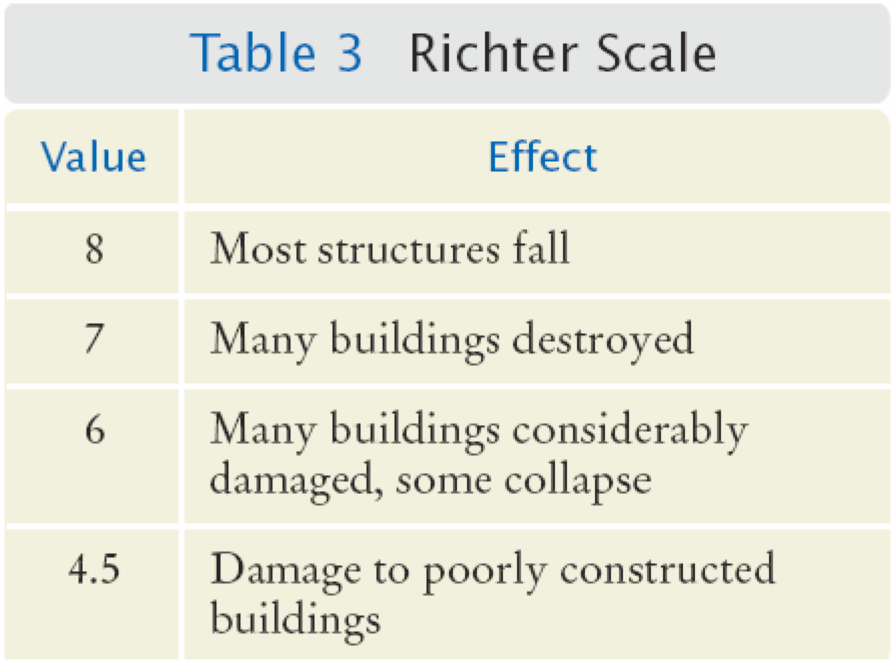


Table 3 Richter Scale

Value	Effect
8	Most structures fall
7	Many buildings destroyed
6	Many buildings considerably damaged, some collapse
4.5	Damage to poorly constructed buildings

When using multiple `if` statements, test general conditions **after** more specific conditions.

if, else if multiway branching

```
if (richter >= 8.0)   // Handle the ‘special case’ first
{
  System.out.println("Most structures fall");
}
else if (richter >= 7.0)
{
  System.out.println("Many buildings destroyed");
}
else if (richter >= 6.0)
{
  System.out.println("Many buildings damaged, some collapse");
}
else if (richter >= 4.5)
{
  System.out.println("Damage to poorly constructed buildings");
}
else    // so that the ‘general case’ can be handled last
{
  System.out.println("No destruction of buildings");
}
```

What is wrong with this code?

```
if (richter >= 8.0)
{
  System.out.println("Most structures fall");
}
if (richter >= 7.0)
{
  System.out.println("Many buildings destroyed");
}
if (richter >= 6.0)
{
  System.out.println("Many buildings damaged, some collapse");
}
if (richter >= 4.5)
{
  System.out.println("Damage to poorly constructed buildings");
}
```

Another way to multiway branch

- The switch statement chooses a case based on an **integer** value.
- break ends each case
- default catches all other values

```
int digit = . . .;
switch (digit)
{
   case 1: digitName = "one";    break;
   case 2: digitName = "two";    break;
   case 3: digitName = "three";  break;
   case 4: digitName = "four";   break;
   case 5: digitName = "five";   break;
   case 6: digitName = "six";    break;
   case 7: digitName = "seven";  break;
   case 8: digitName = "eight";  break;
   case 9: digitName = "nine";   break;
   default: digitName = "";      break;
}
```

If the break is missing, the case *falls through* to the next case’s statements.

Tax Example: Nested `if`s

- Four outcomes (branches)
  - Single
    - <= 32000
    - > 32000
  - Married
    - <= 64000
    - > 64000

Table 4 Federal Tax Rate Schedule

If your status is Single and if the taxable income is	the tax is	of the amount over
at most $32,000	10%	$0
over $32,000	$3,200 + 25%	$32,000
If your status is Married and if the taxable income is	**the tax is**	**of the amount over**
at most $64,000	10%	$0
over $64,000	$6,400 + 25%	$64,000

Common Error 3.4

The Dangling else Problem

- When an if statement is nested inside another if statement, the following can occur:

```
double shippingCharge = 5.00; // $5 inside continental U.S.
if (country.equals("USA"))
  if (state.equals("HI"))
    shippingCharge = 10.00;    // Hawaii is more expensive
else // Pitfall!
  shippingCharge = 20.00;      // As are foreign shipment
```

- The indentation level suggests that the else is related to the if country (“USA”)
  - Else clauses always associate to the closest if

Enumerated Types

- Name a finite list of values that a variable can hold
  - It is like declaring a new type, with a list of possible values

```
public enum FilingStatus {
 SINGLE, MARRIED,MARRIED_FILING_SEPARATELY }
```

  - You can have any number of values, but you must include them all in the enum declaration

  - You can declare variables of the enumeration type:

```
FilingStatus status = FilingStatus.SINGLE;
```

  - And you can use the comparison operator with them:

```
if (status == FilingStatus.SINGLE) . . .
```

3.7 Boolean Variables

- Boolean Variables
  - A Boolean variable is often called a flag because it can be either up (`true`) or down (`false`)
  - `boolean` is a Java data type
    - `boolean failed = true;`
    - Can be either `true` or `false`
- Boolean Operators: && and ||
  - They combine multiple conditions
  - && is the *and* operator
  - || is the *or* operator

Combined Conditions: &&

- Combining two conditions is often used in range checking
  - Is a value between two other values?
- Both sides of the *and* must be true for the result to be true

```
if (temp > 0 && temp < 100)
{
  System.out.println("Liquid");
}
```

A	B	A && B
true	true	true
true	false	false
false	true	false
false	false	false

Combined Conditions: ||

- If only one of two conditions need to be true
  - Use a compound conditional with an or:

```
if (balance > 100 || credit > 100)
{
  System.out.println(“Accepted");
}
```

- If either is true
  - The result is true

A	B	A || B
true	true	true
true	false	true
false	true	true
false	false	false

The *not* Operator: !

- If you need to invert a boolean variable or

```
if (!attending || grade < 60)
{
  System.out.println(“Drop?");
}
```

```
if (attending && !(grade < 60))
{
  System.out.println(“Stay");
}
```

A	!A
true	false
false	true

- If using !, try to use simpler logic:

```
if (attending && (grade >= 60))
```

Range Checking

```
if (temp > 0 && temp < 100)
{
  System.out.println("Liquid");
}
```

- This is often called 'range checking'
  - Used to validate that input is between two values

Another form of ‘range checking’

- Another form of ‘range checking’
  - Checks if value is outside a range

```
if (temp <= 0 || temp >= 100)
{
  System.out.println(“Not Liquid");
}
```

Common Error

- Combining Multiple Relational Operators

```
if (0 <= temp <= 100)  // Syntax error!
```

  - This format is used in math, but not in Java!
  - It requires two comparisons:

```
if (0 <= temp && temp <= 100)
```

- This is also not allowed in Java:

```
if (input == 1 || 2)  // Syntax error!
```

  - This also requires two comparisons:

```
if (input == 1 || input == 2)
```

Short-Circuit Evaluation: &&

- Combined conditions are evaluated from left to right
  - If the left half of an *and* condition is false, why look further?

```
if (temp > 0 && temp < 100)
{
  System.out.println("Liquid");
}
```

- A useful example:

```
if (quantity > 0 && price / quantity < 10)
```

Short-Circuit Evaluation: ||

- If the left half of the *or* is true, why look further?

```
if (temp <= 0 || temp >= 100)
{
  System.out.println(“Not Liquid");
}
```

- Java doesn’t!
- Don’t do these second in the condition:
  - Assignment
  - Output

ElevatorSimulation

- See code

Character Class

Character Class

- The `Character` class has a number of handy methods that return a `boolean` value:

```
if (Character.isDigit(ch))
{
   ...
}
```

Table 5 Character Testing Methods

Method	Examples of Accepted Characters
isDigit	0, 1, 2
isLetter	A, B, C, a, b, c
isUpperCase	A, B, C
isLowerCase	a, b, c
isWhiteSpace	space, newline, tab