

# CHAPTER

# 3

# DECISIONS





# Chapter Goals

- ❑ To implement decisions using the `if` statement
- ❑ To compare integers, floating-point numbers, and Strings
- ❑ To write statements using the Boolean data type
- ❑ To develop strategies for testing your programs
- ❑ To validate user input

In this chapter, you will learn how to program simple and complex decisions. You will apply what you learn to the task of checking user input.



# Contents

- ❑ The **if** Statement
- ❑ Comparing Numbers and Strings
- ❑ Multiple Alternatives
- ❑ Nested Branches
- ❑ Problem Solving: Flowcharts
- ❑ Problem Solving: Test Cases
- ❑ Boolean Variables and Operators
- ❑ Application: Input Validation





## 3.1 The if Statement

- A computer program often needs to make decisions based on input, or circumstances
- For example, buildings often ‘skip’ the 13<sup>th</sup> floor, and elevators should too
  - The 14<sup>th</sup> floor is really the 13<sup>th</sup> floor
  - So every floor above 12 is really ‘floor - 1’
    - If floor > 12, Actual floor = floor - 1
- The two keywords of the if statement are:
  - **if**
  - **else**

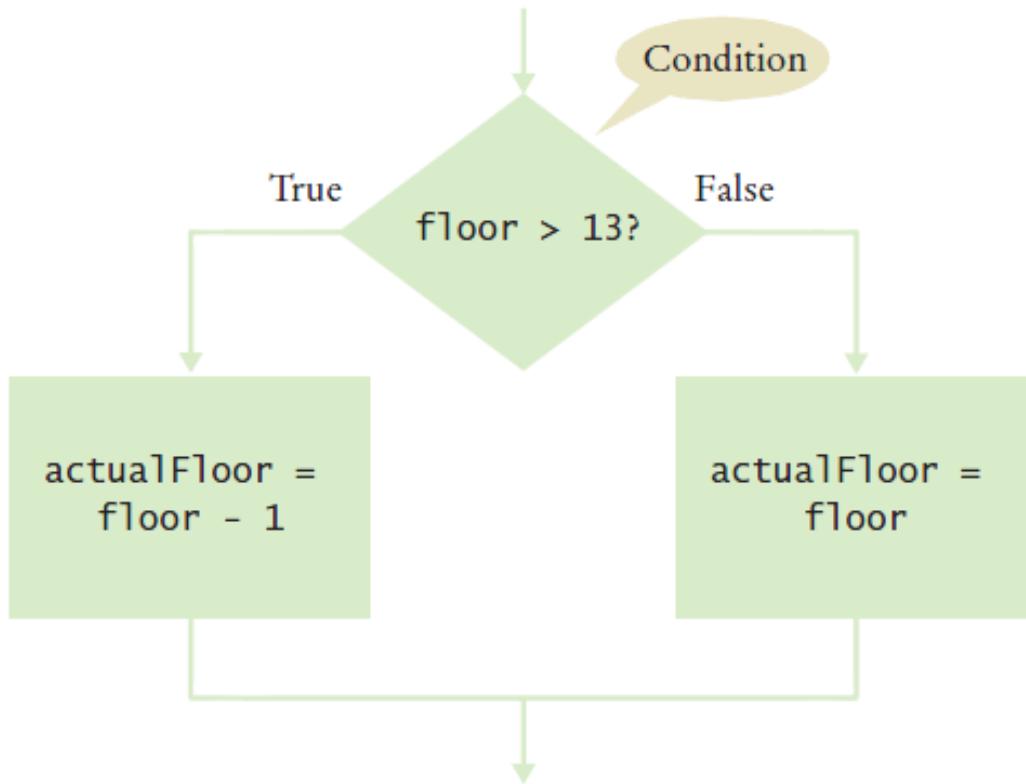


The **if** statement allows a program to carry out different actions depending on the nature of the data to be processed.



# Flowchart of the **if** statement

- One of the two branches is executed once
  - True (**if**) branch      or      False (**else**) branch

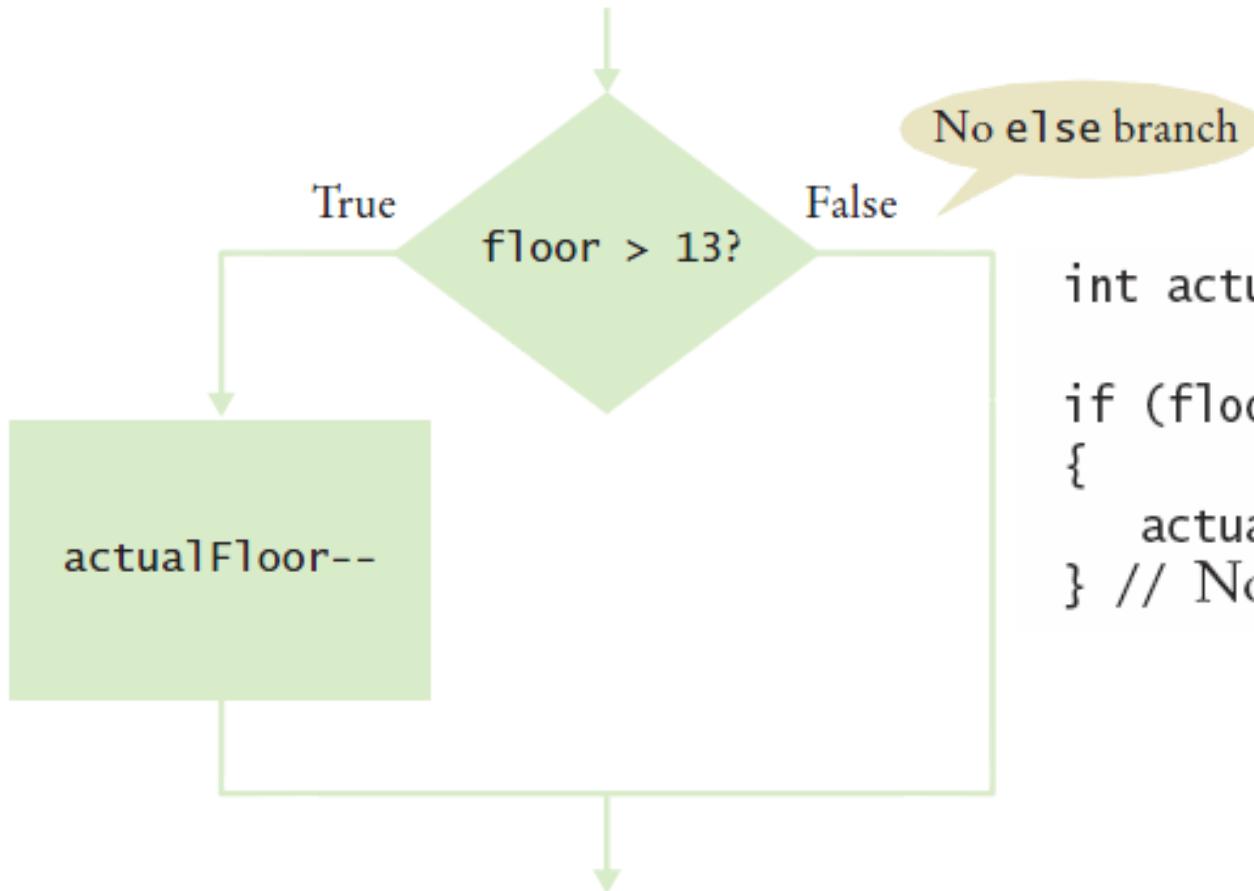


```
int actualFloor;  
  
if (floor > 13)  
{  
    actualFloor = floor - 1;  
}  
else  
{  
    actualFloor = floor;  
}
```



# Flowchart with only true branch

- An **if** statement may not need a ‘False’ (**else**) branch



```
int actualFloor = floor;  
  
if (floor > 13)  
{  
    actualFloor--;  
} // No else needed
```



# Syntax 3.1: The **if** statement

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

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

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

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

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.



# ElevatorSimulation.java

```
1 import java.util.Scanner;  
2  
3 /**  
4     This program simulates an elevator panel that skips the 13th floor.  
5 */  
6 public class ElevatorSimulation  
7 {  
8     public static void main(String[] args)  
9     {  
10         Scanner in = new Scanner(System.in);  
11         System.out.print("Floor: ");  
12         int floor = in.nextInt();  
13  
14         // Adjust floor if necessary  
15  
16         int actualFloor;  
17         if (floor > 13)  
18         {  
19             actualFloor = floor - 1;  
20         }  
21         else  
22         {  
23             actualFloor = floor;  
24         }  
25  
26         System.out.println("The elevator will travel to the actual floor " +  
27             actualFloor);  
28     }  
29 }
```

## Program Run

```
Floor: 20  
The elevator will travel to the actual floor 19
```



# Tips On Using Braces

## □ Line up all pairs of braces vertically

- Lined up

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

- Not aligned (saves lines)

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

## □ Always use braces

- Although single statement clauses do not require them

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

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

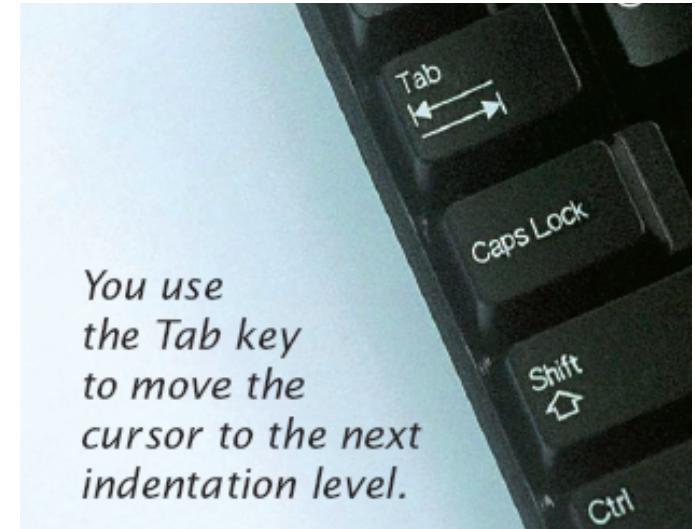
Most programmer's editors have a tool to align matching braces.



# 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 3.1



### 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--;  
}
```

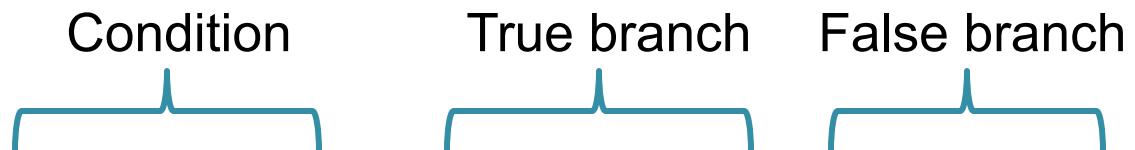


- The ‘body’ (between the curly braces) will always be executed in this case



# The Conditional Operator

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



```
actualFloor = floor > 13 ? floor - 1 : floor;
```

- 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



## 3.2 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
>=	≥	Greater than or equal
<	<	Less than
<=	≤	Less than or equal
==	=	Equal
!=	≠	Not equal



# Syntax 3.2: Comparisons

Check that you have the right direction:  
`>` (greater) or `<` (less)

Check the boundary condition:  
`>` (greater) or `>=` (greater or equal)?

Use `==`, not `=`.

These quantities are compared.

`floor > 13`

One of: `==` `!=` `<` `<=` `>` `>=` (See page 89.)

`floor == 13`

Checks for equality.

```
String input;  
if (input.equals("Y"))
```

Use `equals` to compare strings. (See page 92.)

```
double x; double y; final double EPSILON = 1E-14;  
if (Math.abs(x - y) < EPSILON)
```



Checks that these floating-point numbers are very close.  
See page 91.





# 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

Calculations



```
actualFloor = floor + 1;
```

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



# Relational Operator Use (1)

Table 2 Relational Operator Examples

Expression	Value	Comment
$3 \leq 4$	true	$3$ is less than $4$ ; $\leq$ tests for “less than or equal”.
$3 =< 4$	Error	The “less than or equal” operator is $\leq$ , not $=<$ . The “less than” symbol comes first.
$3 > 4$	false	$>$ is the opposite of $\leq$ .
$4 < 4$	false	The left-hand side must be strictly smaller than the right-hand side.
$4 \leq 4$	true	Both sides are equal; $\leq$ 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

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

- ❑ Strings are a bit ‘special’ in Java
- ❑ Do not use the `==` operator with Strings
  - The following compares the locations of two strings, and not their contents

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

- ❑ Instead use the String’s `equals` method:

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



## Common Error 3.2



- ❑ Comparison of Floating-Point Numbers
  - Floating-point numbers have limited precision
  - Round-off errors can lead to unexpected results

```
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 2.0000000000000044



# 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");
}
```



# Common Error 3.3



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

```
String nickname = "Rob";  
.  
.  
.  
if (nickname == "Rob") // Test is true
```

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



# 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



# Implementing an **if** Statement

- 1) Decide on a branching condition

original price < 128?

- 2) Write pseudocode for the true branch

discounted price = 0.92 x original price

- 3) Write pseudocode for the false branch

discounted price = 0.84 x original price

- 4) Double-check relational operators

- Test values below, at, and above the comparison (127, 128, 129)



## Implementing an **if** Statement (cont.)

- 5) Remove duplication

discounted price = \_\_\_\_ x original price

- 6) Test both branches

discounted price =  $0.92 \times 100 = 92$

discounted price =  $0.84 \times 200 = 168$

- 7) Write the code in Java



# Implemented Example

- The university bookstore has a Kilobyte Day sale every October 24, giving an 8 percent discount on all computer accessory purchases if the price is less than \$128, and a 16 percent discount if the price is at least \$128.

```
if (originalPrice < 128)
{
    discountRate = 0.92;
}
else
{
    discountRate = 0.84;
}
discountedPrice = discountRate * originalPrice;
```



## 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

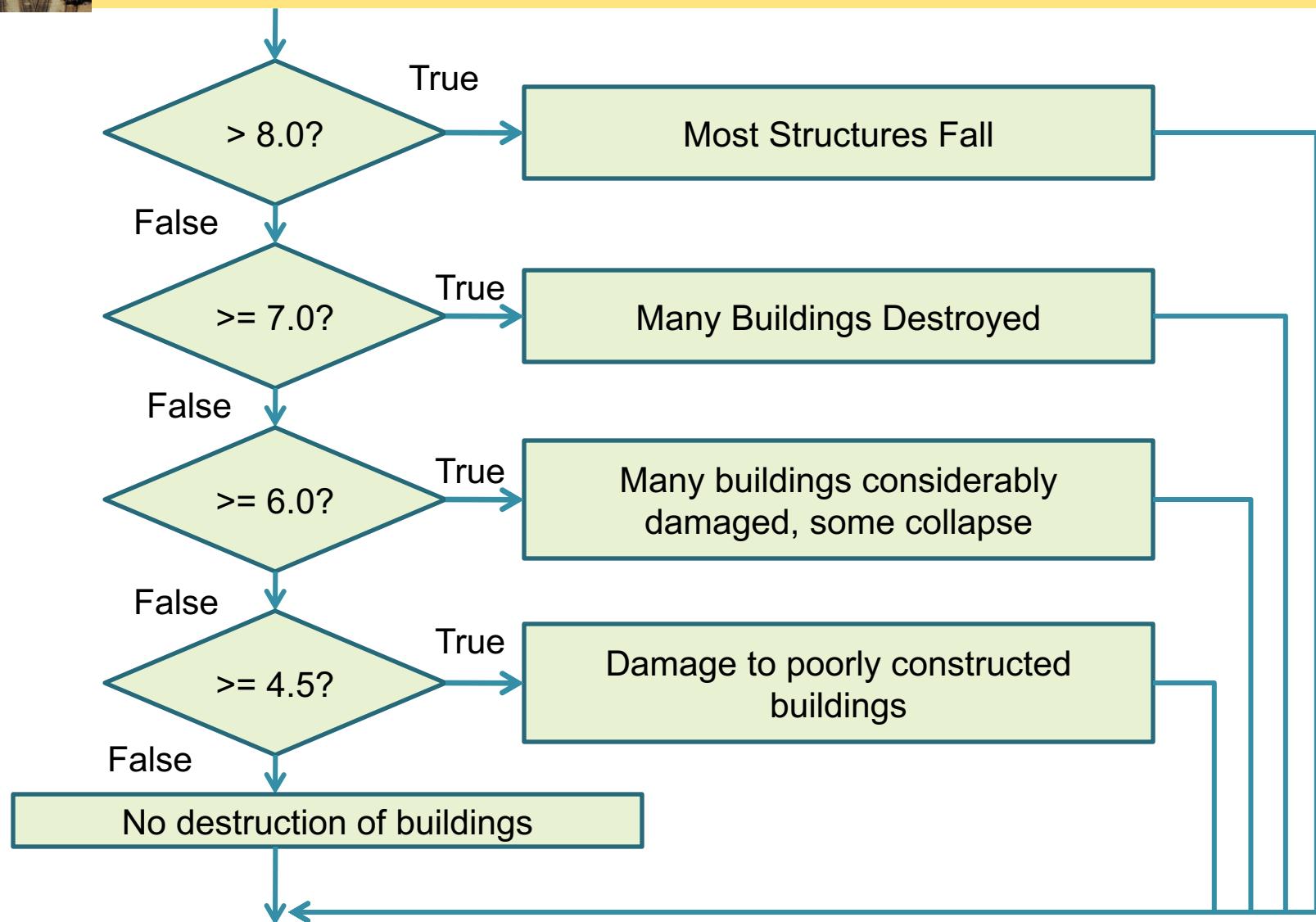
When using multiple **if** statements, test general conditions after more specific conditions.

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



# Flowchart of Multiway branching





# 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

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

```
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;
}
```

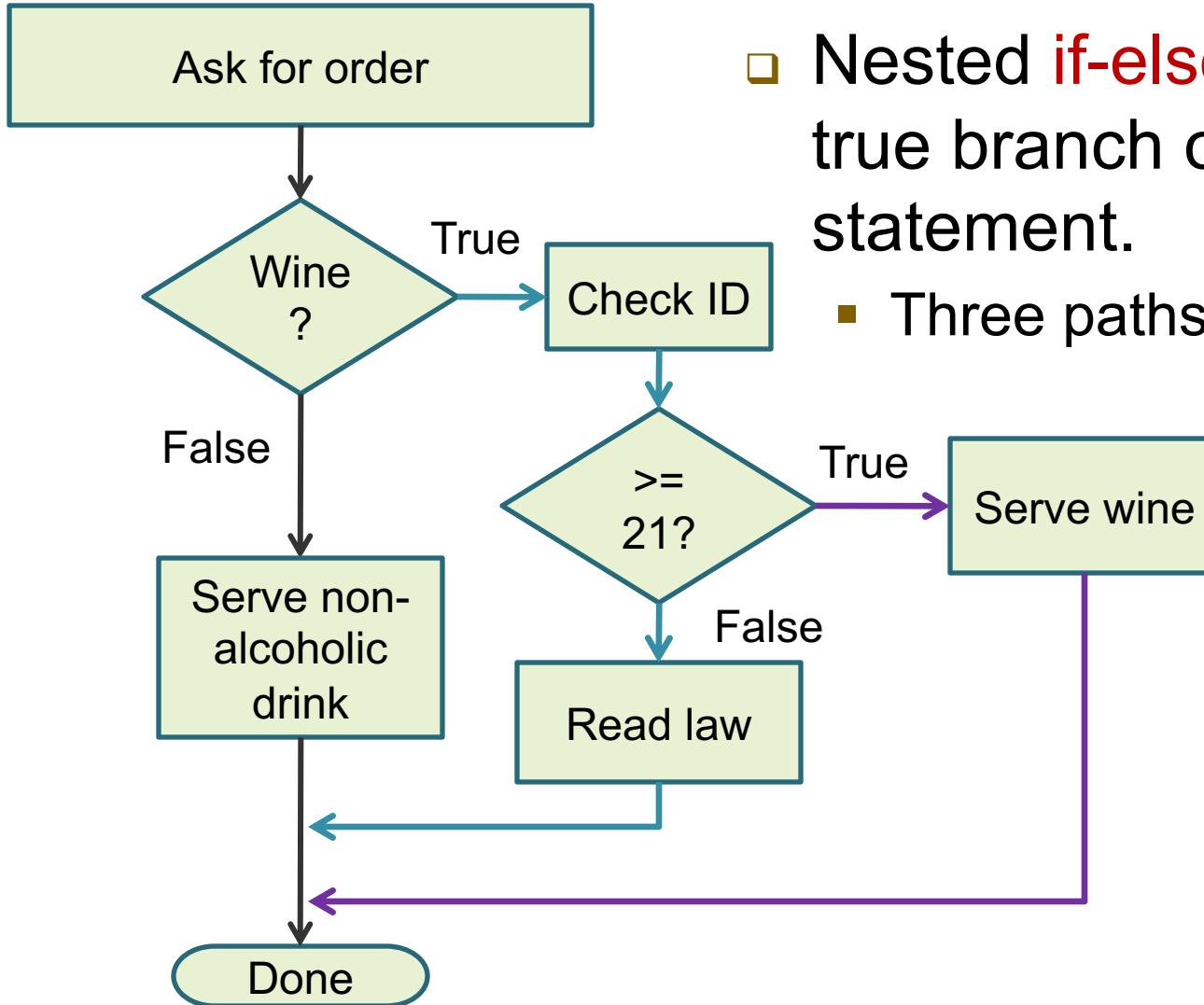


## 3.4 Nested Branches

- ❑ You can *nest* an **if** inside either branch of an **if** statement.
- ❑ Simple example: Ordering drinks
  - Ask the customer for their drink order
  - **if** customer orders wine
    - Ask customer for ID
    - **if** customer's age is 21 or over
      - Serve wine
    - Else
      - Politely explain the law to the customer
  - Else
    - Serve customers a non-alcoholic drink



# Flowchart of a Nested if



- Nested **if-else** inside true branch of an **if** statement.
  - Three paths



# Tax Example: Nested ifs

- Four outcomes (branches)

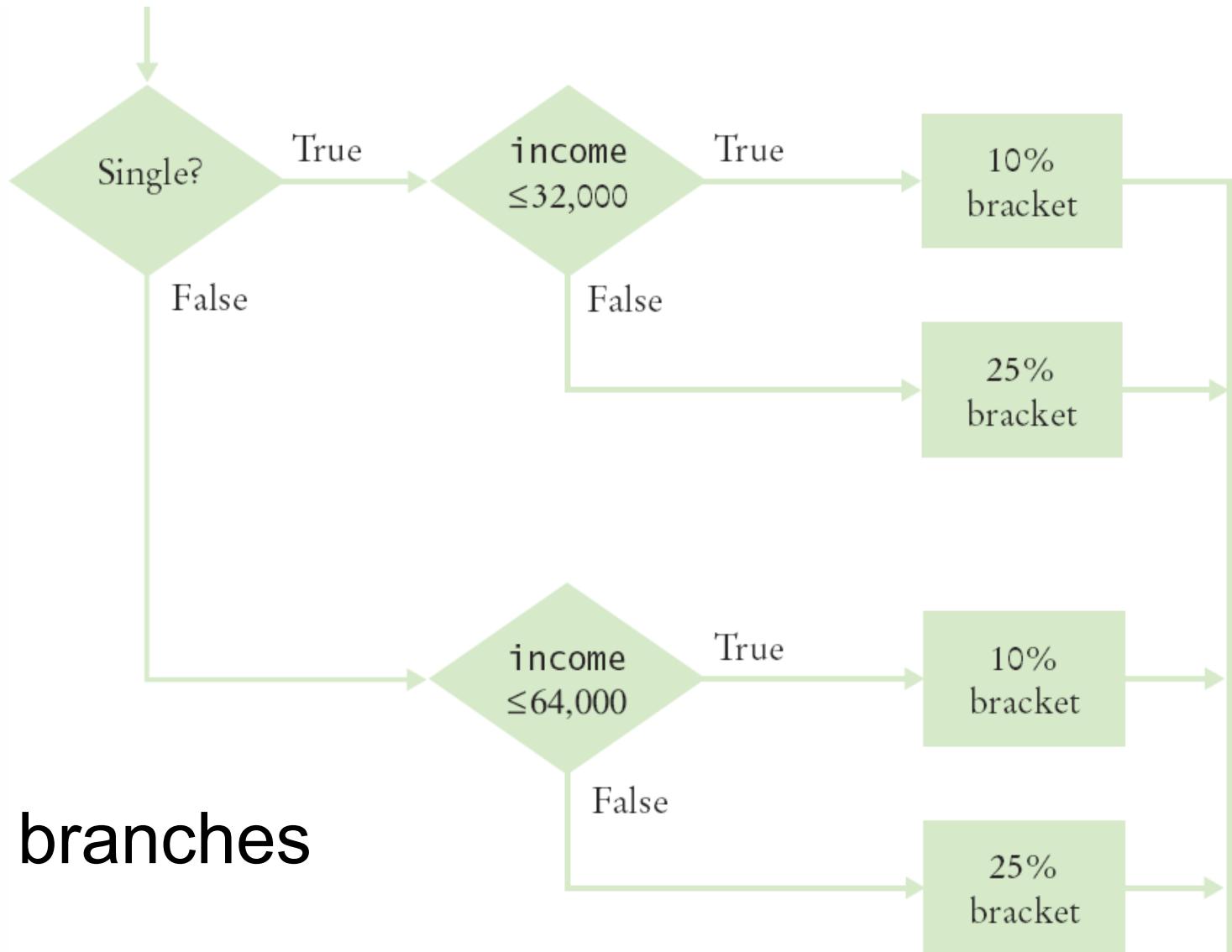
Table 4 Federal Tax Rate Schedule

- Single
  - $\leq 32000$
  - $> 32000$
- Married
  - $\leq 64000$
  - $> 64000$

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



# Flowchart for Tax Example



□ Four branches



# TaxCalculator.java (1)

```
1 import java.util.Scanner;
2
3 /**
4  * This program computes income taxes, using a simplified tax schedule.
5 */
6 public class TaxCalculator
7 {
8     public static void main(String[] args)
9     {
10         final double RATE1 = 0.10;
11         final double RATE2 = 0.25;
12         final double RATE1_SINGLE_LIMIT = 32000;
13         final double RATE1_MARRIED_LIMIT = 64000;
14
15         double tax1 = 0;
16         double tax2 = 0;
17
18         // Read income and marital status
19
20         Scanner in = new Scanner(System.in);
21         System.out.print("Please enter your income: ");
22         double income = in.nextDouble();
23
24         System.out.print("Please enter s for single, m for married: ");
25         String maritalStatus = in.next();
```



# TaxCalculator.java (2)

- The ‘True’ branch (Married)
  - Two branches within this branch

```
27 // Compute taxes due
28
29     if (maritalStatus.equals("s"))
30     {
31         if (income <= RATE1_SINGLE_LIMIT)
32         {
33             tax1 = RATE1 * income;
34         }
35         else
36         {
37             tax1 = RATE1 * RATE1_SINGLE_LIMIT;
38             tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT);
39         }
40     }
```



# TaxCalculator.java (3)

## □ The ‘False’ branch (not Married)

```
41 else
42 {
43     if (income <= RATE1_MARRIED_LIMIT)
44     {
45         tax1 = RATE1 * income;
46     }
47     else
48     {
49         tax1 = RATE1 * RATE1_MARRIED_LIMIT;
50         tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
51     }
52 }
53
54 double totalTax = tax1 + tax2;
55
56 System.out.println("The tax is $" + totalTax);
57 }
58 }
```

### Program Run

```
Please enter your income: 80000
Please enter s for single, m for married: m
The tax is $10400
```



# Hand-Tracing



- ❑ Hand-tracing helps you understand whether a program works correctly
- ❑ Create a table of key variables
  - Use pencil and paper to track their values
- ❑ Works with pseudocode or code
  - Track location with a marker such as a paper clip
- ❑ Use example input values that:
  - You know what the correct outcome should be
  - Will test each branch of your code





# Hand-Tracing Tax Example (1)

tax1	tax2	income	marital status
0	0		

## □ Setup

- Table of variables
- Initial values

```
8 public static void main(String[] args)
9 {
10     final double RATE1 = 0.10;
11     final double RATE2 = 0.25;
12     final double RATE1_SINGLE_LIMIT = 32000;
13     final double RATE1_MARRIED_LIMIT = 64000;
14
15     double tax1 = 0;
16     double tax2 = 0;
17 }
```



# Hand-Tracing Tax Example (2)

tax1	tax2	income	marital status
0	0	80000	m

- Input variables

- From user
- Update table

```
20 Scanner in = new Scanner(System.in);
21 System.out.print("Please enter your income: ");
22 double income = in.nextDouble();
23
24 System.out.print("Please enter s for single, m for married: ");
25 String maritalStatus = in.next();
```

- Because marital status is not “s” we skip to the else on line 41

```
29 if (maritalStatus.equals("s"))
30 {
41   else
42 {
```



## Hand-Tracing Tax Example (3)

- Because income is not  $\leq 64000$ , we move to the else clause on line 47
  - Update variables on lines 49 and 50
  - Use constants

```
43 if (income <= RATE1_MARRIED_LIMIT)
44 {
45     tax1 = RATE1 * income;
46 }
47 else
48 {
49     tax1 = RATE1 * RATE1_MARRIED_LIMIT;
50     tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
51 }
```

tax1	tax2	income	marital status
0	0	80000	m
6400	4000		



# Hand-Tracing Tax Example (4)

tax1	tax2	income	marital status	total tax
0	0	80000	m	
6400	4000			10400

## □ Output

- Calculate
- As expected?

```
54     double totalTax = tax1 + tax2;  
55  
56     System.out.println("The tax is $" + totalTax);  
57 }
```



# 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

- Java provides an easy way to 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;
```

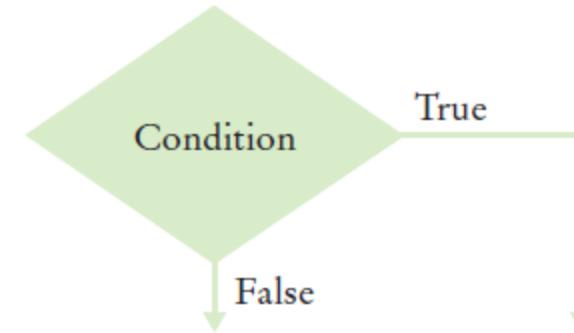
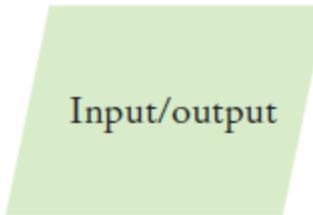
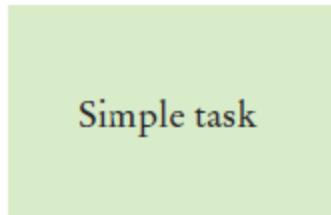
- And you can use the comparison operator with them:

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



## 3.5 Problem Solving: Flowcharts

- ❑ You have seen a few basic flowcharts
- ❑ A flowchart shows the structure of decisions and tasks to solve a problem
- ❑ Basic flowchart elements:



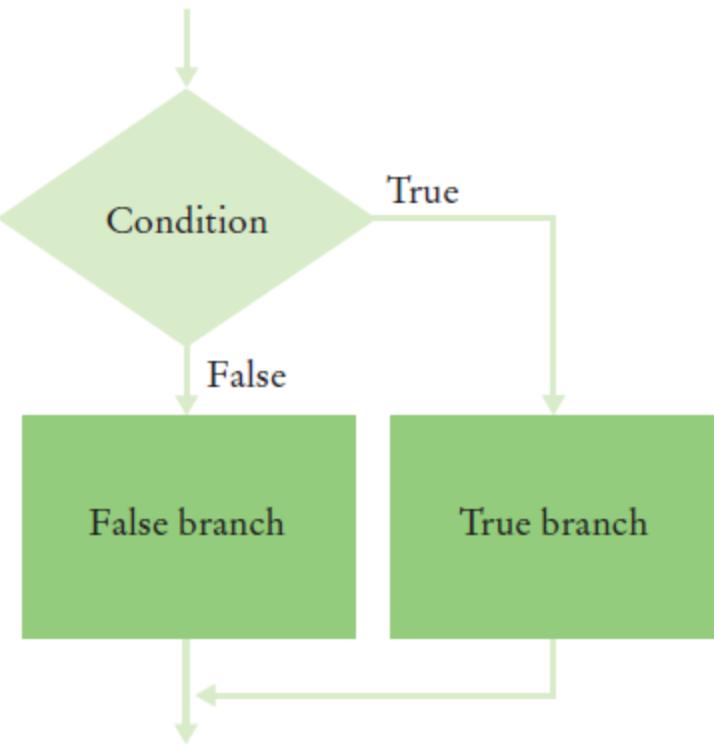
- ❑ Connect them with arrows
  - But never point an arrow inside another branch!

Each branch of a decision can contain tasks and further decisions.

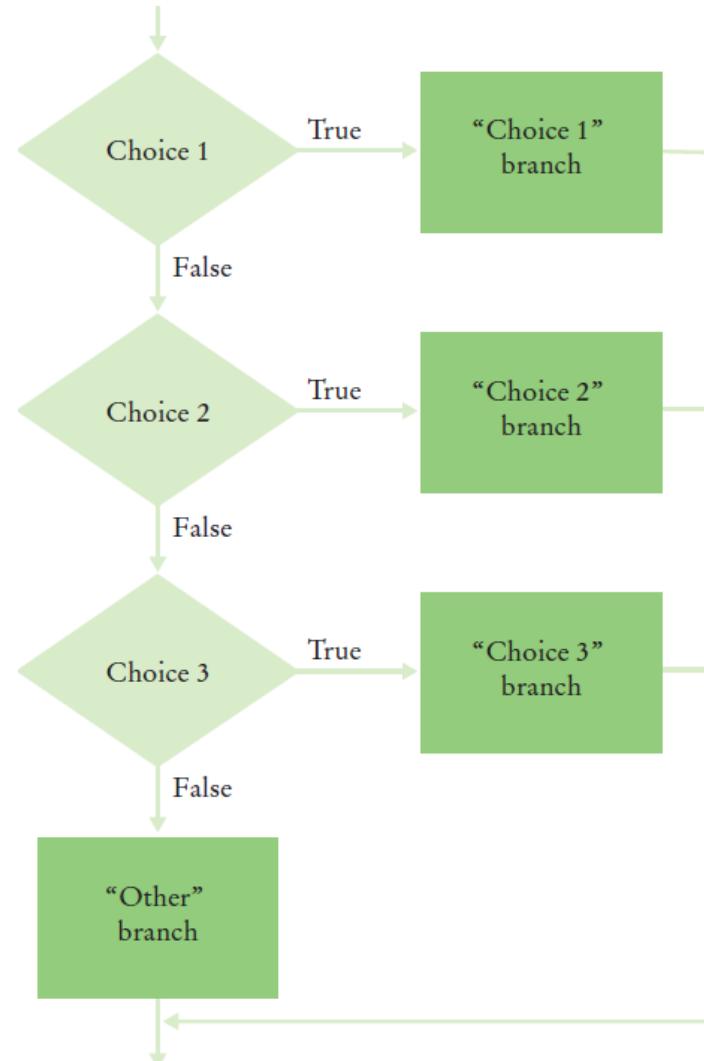


# Conditional Flowcharts

## □ Two Outcomes



## □ Multiple Outcomes

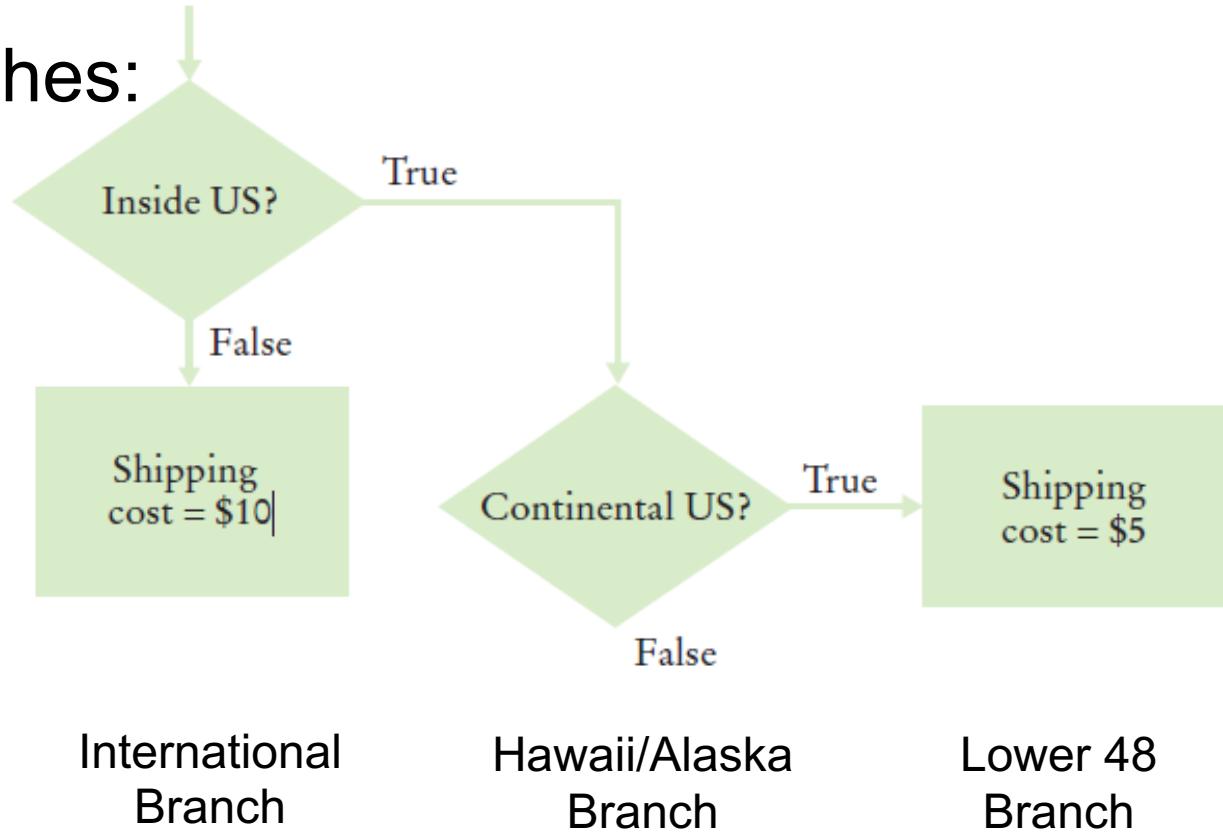




# Shipping Cost Flowchart

Shipping costs are \$5 inside the United States, except that to Hawaii and Alaska they are \$10. International shipping costs are also \$10.

- Three Branches:

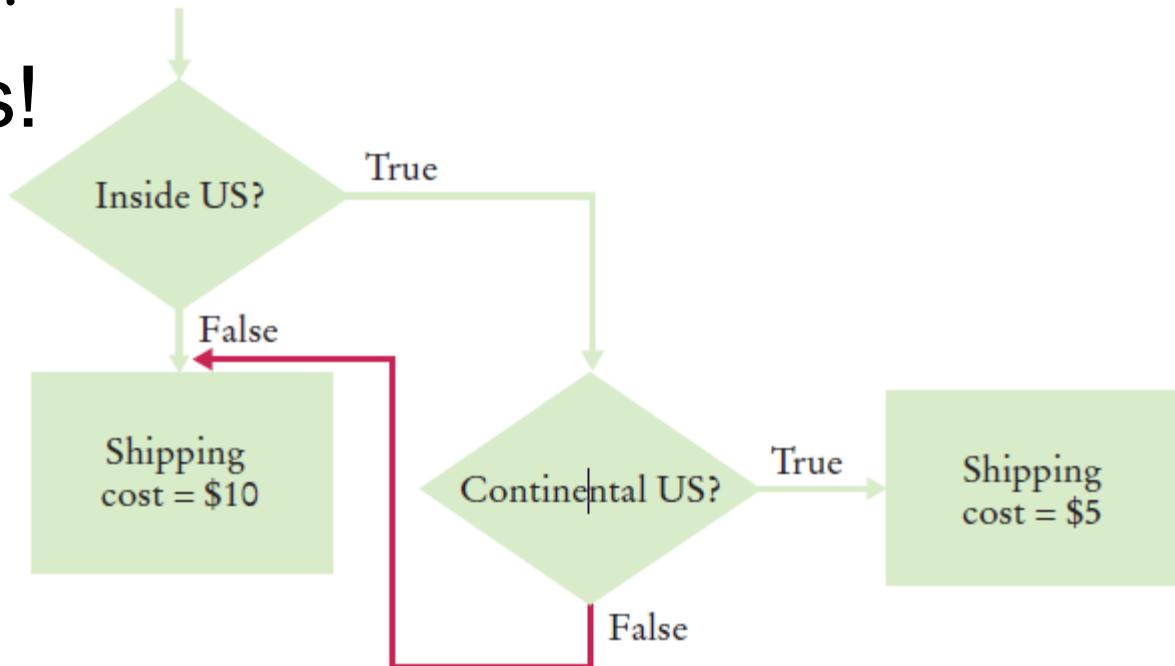




# Don't connect branches!

Shipping costs are \$5 inside the United States, except that to Hawaii and Alaska they are \$10. International shipping costs are also \$10.

- Don't do this!



International  
Branch

Hawaii/Alaska  
Branch

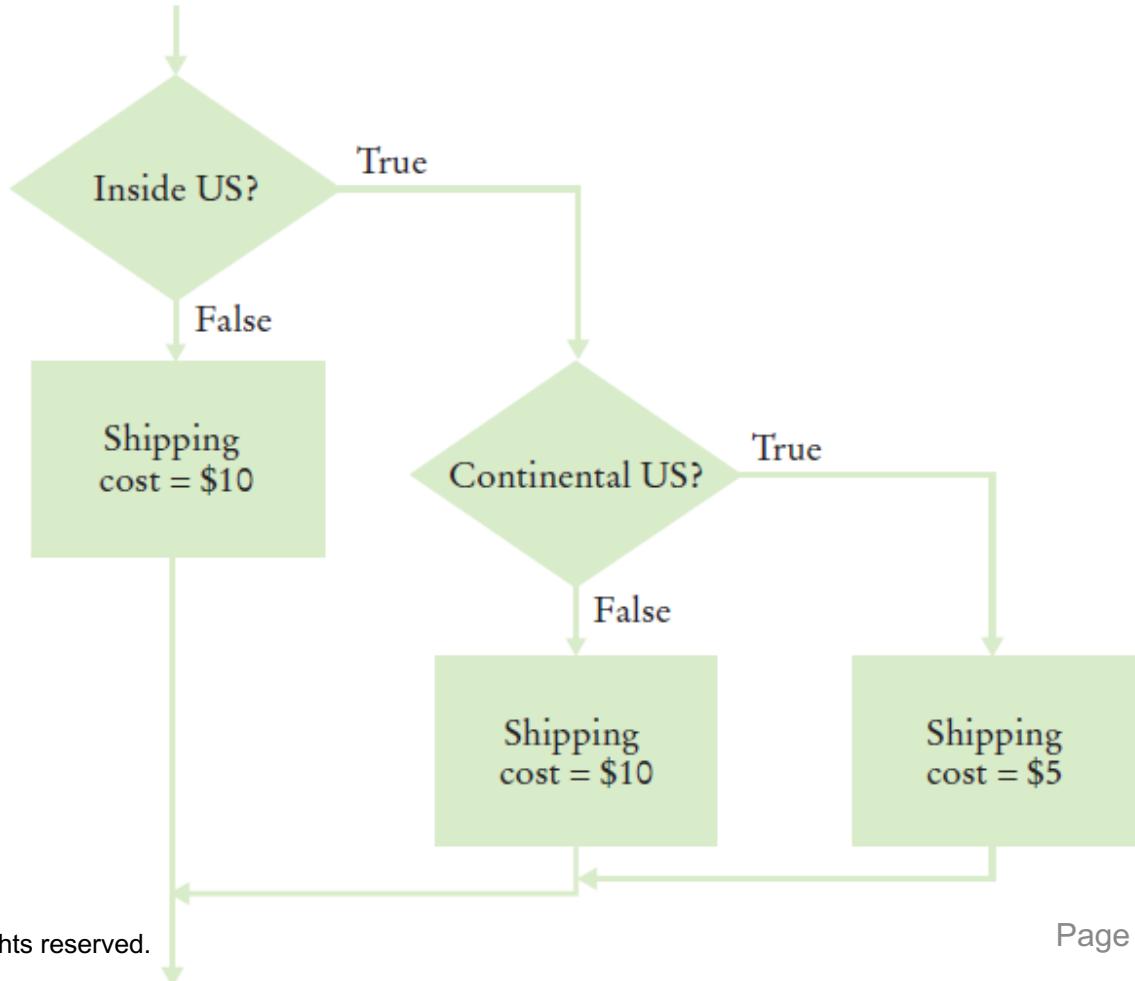
Lower 48  
Branch



# Shipping Cost Flowchart

Shipping costs are \$5 inside the United States, except that to Hawaii and Alaska they are \$10. International shipping costs are also \$10.

- ❑ Completed:





## 3.6 Problem Solving: Test Cases

- ❑ Aim for complete *coverage* of all decision points:
  - There are two possibilities for the marital status and two tax brackets for each status, yielding four test cases
  - Test a handful of *boundary* conditions, such as an income that is at the boundary between two tax brackets, and a zero income
  - If you are responsible for error checking (which is discussed in Section 3.8), also test an invalid input, such as a negative income

Each branch of your code should be covered with a test case



# Choosing Test Cases

## □ Choose input values that:

- Test boundary cases and 0 values
- Test each branch

A *boundary case* is a value that is tested in the code.

Test Case		Expected Output	Comment
30,000	s	3,000	10% bracket
72,000	s	13,200	$3,200 + 25\% \text{ of } 40,000$
50,000	m	5,000	10% bracket
104,000	m	16,400	$6,400 + 25\% \text{ of } 40,000$
32,000	m	3,200	boundary case
0		0	boundary case



## 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





# Character Testing Methods

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

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

## 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



# 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");
}
```

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

- If either is true
  - The result is true



# The *not* Operator: !

- ❑ If you need to invert a boolean variable or comparison, precede it with !

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

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

- ❑ If using !, try to use simpler logic:

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

A	!A
true	false
false	true

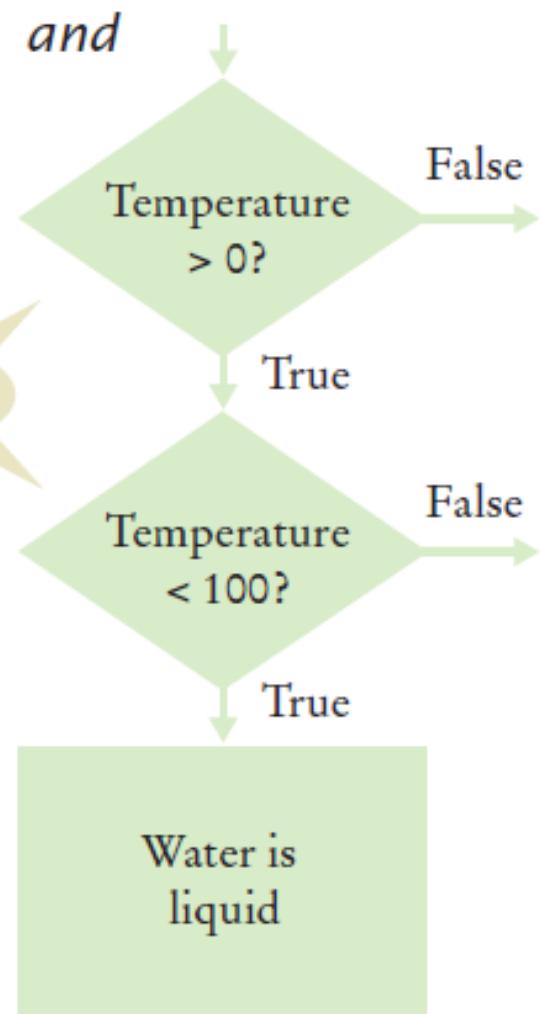


# and Flowchart

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

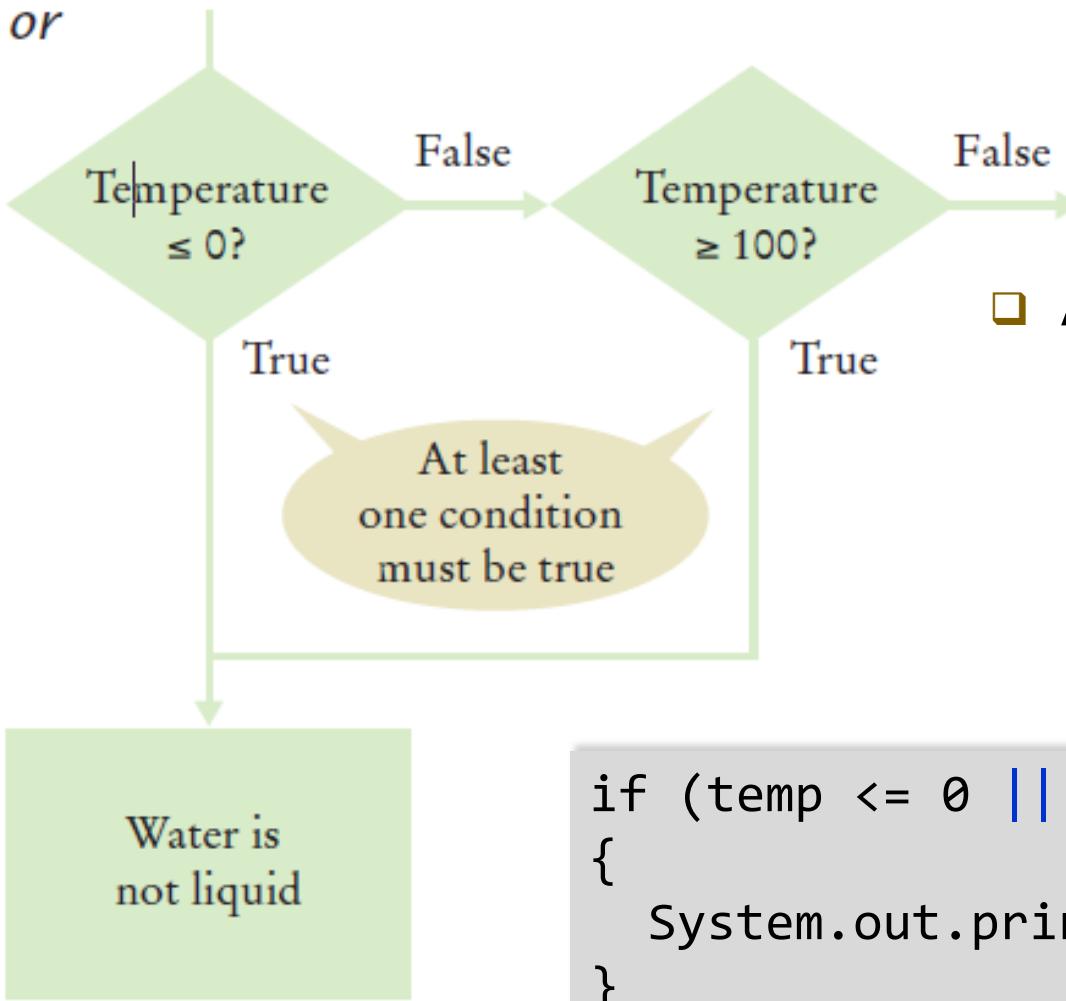
Both conditions  
must be true

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





# or Flowchart



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

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



# Boolean Operator Examples

Table 5 Boolean Operator Examples

Expression	Value	Comment
<code>0 &lt; 200 &amp;&amp; 200 &lt; 100</code>	false	Only the first condition is true.
<code>0 &lt; 200    200 &lt; 100</code>	true	The first condition is true.
<code>0 &lt; 200    100 &lt; 200</code>	true	The <code>  </code> is not a test for “either-or”. If both conditions are true, the result is true.
<code>0 &lt; x &amp;&amp; x &lt; 100    x == -1</code>	<code>(0 &lt; x &amp;&amp; x &lt; 100)    x == -1</code>	The <code>&amp;&amp;</code> operator has a higher precedence than the <code>  </code> operator (see Appendix B).
 <code>0 &lt; x &lt; 100</code>	Error	<b>Error:</b> This expression does not test whether <code>x</code> is between 0 and 100. The expression <code>0 &lt; x</code> is a Boolean value. You cannot compare a Boolean value with the integer 100.



# Boolean Operator Examples

Table 5 Boolean Operator Examples

 <code>x &amp;&amp; y &gt; 0</code>	<b>Error</b>	<b>Error:</b> This expression does not test whether <code>x</code> and <code>y</code> are positive. The left-hand side of <code>&amp;&amp;</code> is an integer, <code>x</code> , and the right-hand side, <code>y &gt; 0</code> , is a Boolean value. You cannot use <code>&amp;&amp;</code> with an integer argument.
<code>!(0 &lt; 200)</code>	<code>false</code>	<code>0 &lt; 200</code> is <code>true</code> , therefore its negation is <code>false</code> .
<code>frozen == true</code>	<code>frozen</code>	There is no need to compare a Boolean variable with <code>true</code> .
<code>frozen == false</code>	<code>!frozen</code>	It is clearer to use <code>!</code> than to compare with <code>false</code> .



# Common Error 3.5



## □ 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)
```



## Common Error 3.6



### Confusing `&&` and `||` Conditions

- It is a surprisingly common error to confuse `&&` and `||` conditions.
- A value lies between 0 and 100 if it is at least 0 *and* at most 100.
- It lies outside that range if it is less than 0 *or* greater than 100.
- There is no golden rule; you just have to think carefully.



# Short-Circuit Evaluation: `&&`

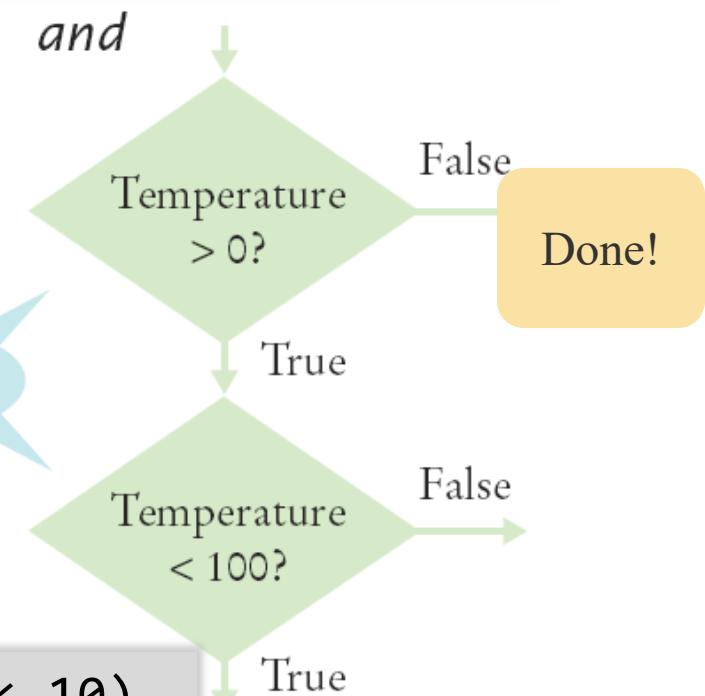
- Combined conditions are evaluated from left to right
  - If the left half of an *and* condition is false, why look further?

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

Both conditions  
must be true

- A useful example:

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



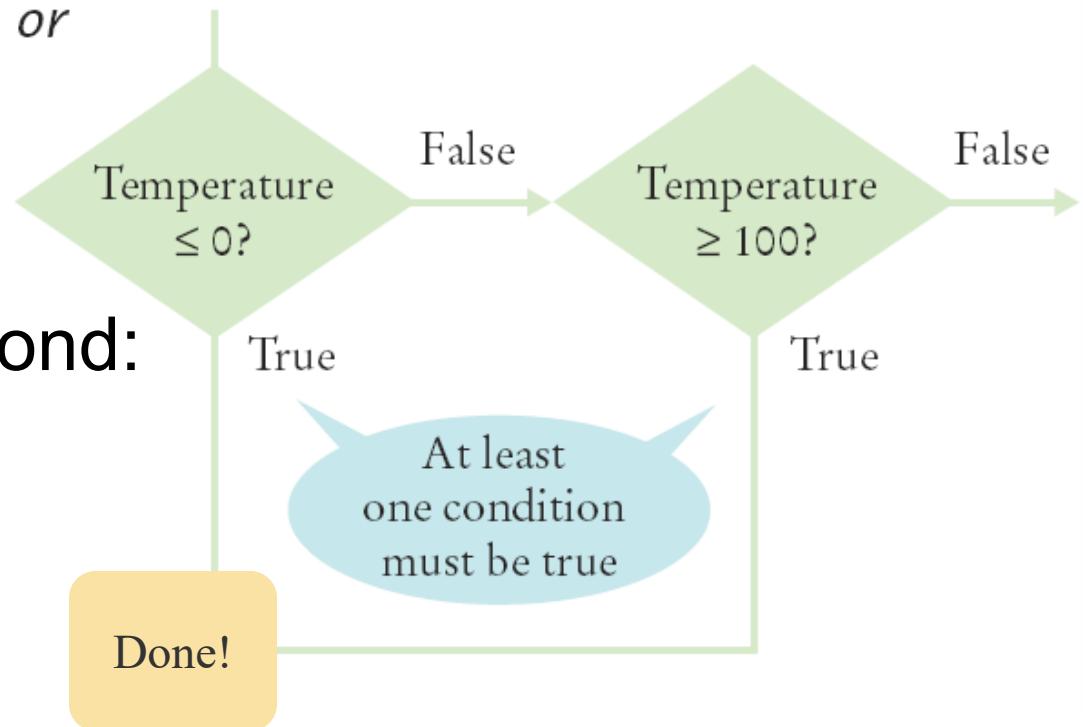


# Short-Circuit Evaluation: ||

- 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:
  - Assignment
  - Output





# De Morgan's Law

- De Morgan's law tells you how to negate **&&** and **||** conditions:
  - $!(A \ \&\& \ B)$       is the same as       $!A \ || \ !B$
  - $!(A \ || \ B)$       is the same as       $!A \ \&\& \ !B$
- Example: Shipping is higher to AK and HI

```
if (!(country.equals("USA")  
    && !state.equals("AK")  
    && !state.equals("HI")))  
    shippingCharge = 20.00;
```

```
if !country.equals("USA")  
    || state.equals("AK")  
    || state.equals("HI")  
    shippingCharge = 20.00;
```

- To simplify conditions with negations of *and* or *or* expressions, it is usually a good idea to apply De Morgan's Law to move the negations to the innermost level.



## 3.8 Input Validation

### □ Accepting user input is dangerous

- Consider the Elevator program:
- The user may input an invalid character or value
- Must be an integer
  - Scanner can help!
  - `hasNextInt`
    - True if integer
    - False if not

```
if (in.hasNextInt())
{
    int floor = in.nextInt();
    // Process the input value
}
else
{
    System.out.println("Not integer.");
}
```

- Then range check value
- We expect a floor number to be between 1 and 20
  - NOT 0, 13 or > 20



# ElevatorSimulation2.java

```
7 public class ElevatorSimulation2
8 {
9     public static void main(String[] args)
10    {
11        Scanner in = new Scanner(System.in);
12        System.out.print("Floor: ");
13        if (!in.hasNextInt())
14        {
15            // Now we know that the user entered an integer
16
17            int floor = in.nextInt();
18
19            if (floor == 13)
20            {
21                System.out.println("Error: There is no thirteenth floor.");
22            }
23            else if (floor <= 0 || floor > 20)      Input value range checking
24            {
25                System.out.println("Error: The floor must be between 1 and 20.");
26            }
27            else
28            {
29                // Now we know that the input is valid
```

Input value validity checking

Input value range checking



# ElevatorSimulation2.java

```
30
31     int actualFloor = floor;
32     if (floor > 13)
33     {
34         actualFloor = floor - 1;
35     }
36
37     System.out.println("The elevator will travel to the actual floor "
38                         + actualFloor);
39 }
40 }
41 else
42 {
43     System.out.println("Error: Not an integer.");
44 }
45 }
46 }
```

## Program Run

Floor: 13

Error: There is no thirteenth floor.



# Summary: **if** Statement

- ❑ The **if** statement allows a program to carry out different actions depending on the nature of the data to be processed.
- ❑ Relational operators (`<` `<=` `>` `>=` `==` `!=`) are used to compare numbers and Strings.
- ❑ Do not use the `==` operator to compare Strings.
  - Use the `equals` method instead.
  - The `compareTo` method compares Strings in lexicographic order.
- ❑ Multiple **if** statements can be combined to evaluate complex decisions.
- ❑ When using multiple **if** statements, test general conditions after more specific conditions.



# Summary: Flowcharts and Testing

- ❑ When a decision statement is contained inside the branch of another decision statement, the statements are *nested*.
- ❑ Nested decisions are required for problems that have two levels of decision making.
- ❑ Flow charts are made up of elements for tasks, input/output, and decisions.
- ❑ Each branch of a decision can contain tasks and further decisions.
- ❑ Never point an arrow inside another branch.
- ❑ Each branch of your program should be covered by a test case.
- ❑ It is a good idea to design test cases before implementing a program.



# Summary: Boolean

- ❑ The Boolean type **boolean** has two values, **true** and **false**.
  - Java has two Boolean operators that combine conditions: **&&** (*and*) and **||** (*or*).
  - To invert a condition, use the **!** (*not*) operator.
  - The **&&** and **||** operators are computed lazily: As soon as the truth value is determined, no further conditions are evaluated.
  - De Morgan's law tells you how to negate **&&** and **||** conditions.
- ❑ You can use `Scanner hasNext` methods to ensure that the data is what you expect.