CHAPTER 3

DECISIONS

Copyright © 2013 by John Wiley & Sons. All rights reserved.

Slides by Donald W. Smith
TechNeTrain.com

Final Draft
Oct 15, 2011

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 for 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.

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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

Copyright © 2013 by John Wiley & Sons. All rights reserved.

3.1 The `if` Statement

- A computer program often needs to make **decisions** based on input, or circumstances
- For example, buildings often 'skip' the 13th floor, and elevators should too
  - The 14th floor is really the 13th floor
  - So every floor above 12 is really 'floor - 1'
    - If floor > 12, Actual floor = floor - 1

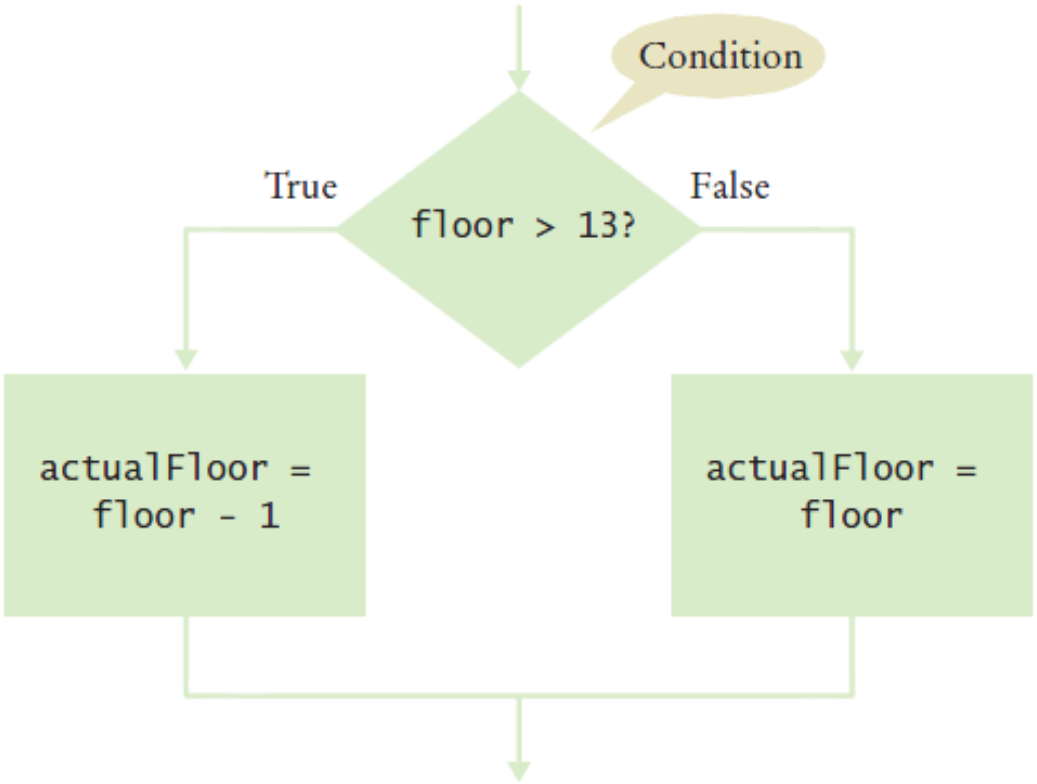
- The two keywords of the if statement are:
  - `if`
  - `else`

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

Copyright © 2013 by John Wiley & Sons. All rights reserved.

Flowchart of the `if` statement

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



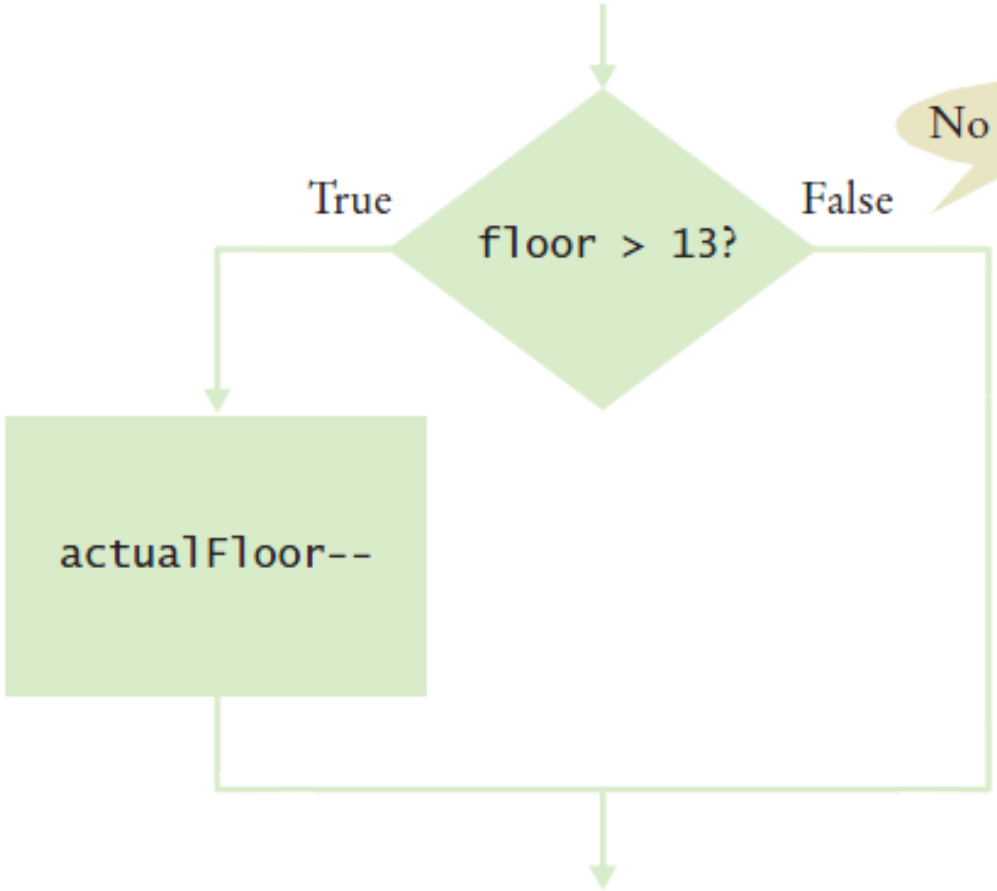

```
int actualFloor;

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

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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
```

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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.

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

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.

Omit the `else` branch if there is nothing to do.

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

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

if (condition)
{
statement
}

if (condition) {statement1}
else {statement2}

If a semicolon after the if condition:
The statement enclosed in braces is no longer a part of the if statement.
It is always executed.
The actualFloor value will be decremented

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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
```

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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--;
```

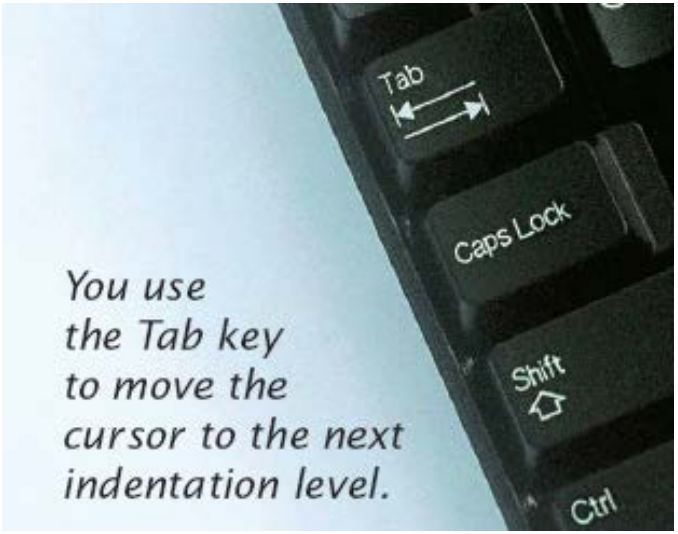
Most programmer's editors have a tool to align matching braces.

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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.

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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

Copyright © 2013 by John Wiley & Sons. All rights reserved.

The Conditional Operator

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

condition ? value1 : value2

Condition — True branch — False branch

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

Equivalent to
if(floor > 13) {actualFloor = floor -1} else {actualFloor = 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

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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

Copyright © 2013 by John Wiley & Sons. All rights reserved.

Syntax 3.2: Comparisons

These quantities are compared.

```
floor > 13
```

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

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

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

```
floor == 13
```

Checks for equality.

Use ==, not =.

= assignment
== relational operator

```
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.

Ex.
floor = 13; //assign 13 to floor

if(floor == 13) // test whether floor equals 13

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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)
```

Ex.
floor – 1 < 13
Equivalent to
(floor -1) < 13

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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.

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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.

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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);
}
```

roundoff errors

```
Output:
Math.sqrt(2.0) squared is not 2.0 but 2.0000000000000044
```

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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");
}
```

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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)) ...
```

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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
```

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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

Copyright © 2013 by John Wiley & Sons. All rights reserved.

Comparing Objects

- `==` tests for identity, `equals` for identical content
- ```
  Rectangle box1 = new Rectangle(5, 10, 20, 30);
  Rectangle box2 = box1;
  Rectangle box3 = new Rectangle(5, 10, 20, 30);
  ```
- `box1 != box3`, but `box1.equals(box3)`
- `box1 == box2`
- Caveat: `equals` must be defined for the class

Rectangle(int x, int y, int width, int height)

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.
  ```

Object Comparison

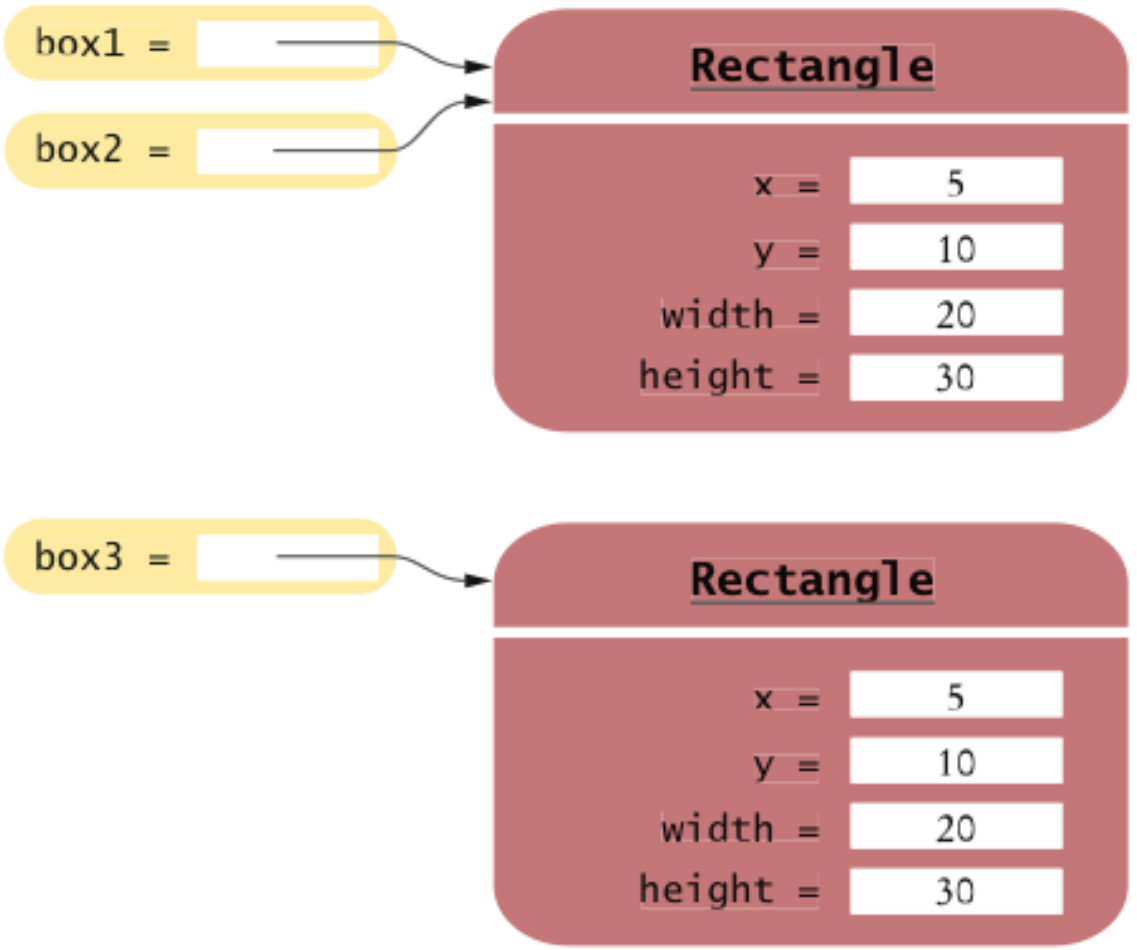


Figure 4
Comparing Object References

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Testing for `null`

- `null` reference refers to no object:

```
String middleInitial = null; // Not set
if ( ... )
   middleInitial = middleName.substring(0, 1);
```

- Can be used in tests:

```
if (middleInitial == null)
   System.out.println(firstName + " " + lastName);
else
   System.out.println(firstName + " " + middleInitial +
      ". " + lastName);
```

- Use `==`, not `equals`, to test for `null`
- `null` is not the same as the empty string `""`

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

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)

Copyright © 2013 by John Wiley & Sons. All rights reserved.

Implementing an `if` Statement (cont.)

5) Remove duplication

discounted price = ____ x original price

6) Test both branches

discounted price = 0.92 x 100 = 92

discounted price = 0.84 x 200 = 168

7) Write the code in Java

Copyright © 2011 by John Wiley & Sons. All rights reserved.

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;
```

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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

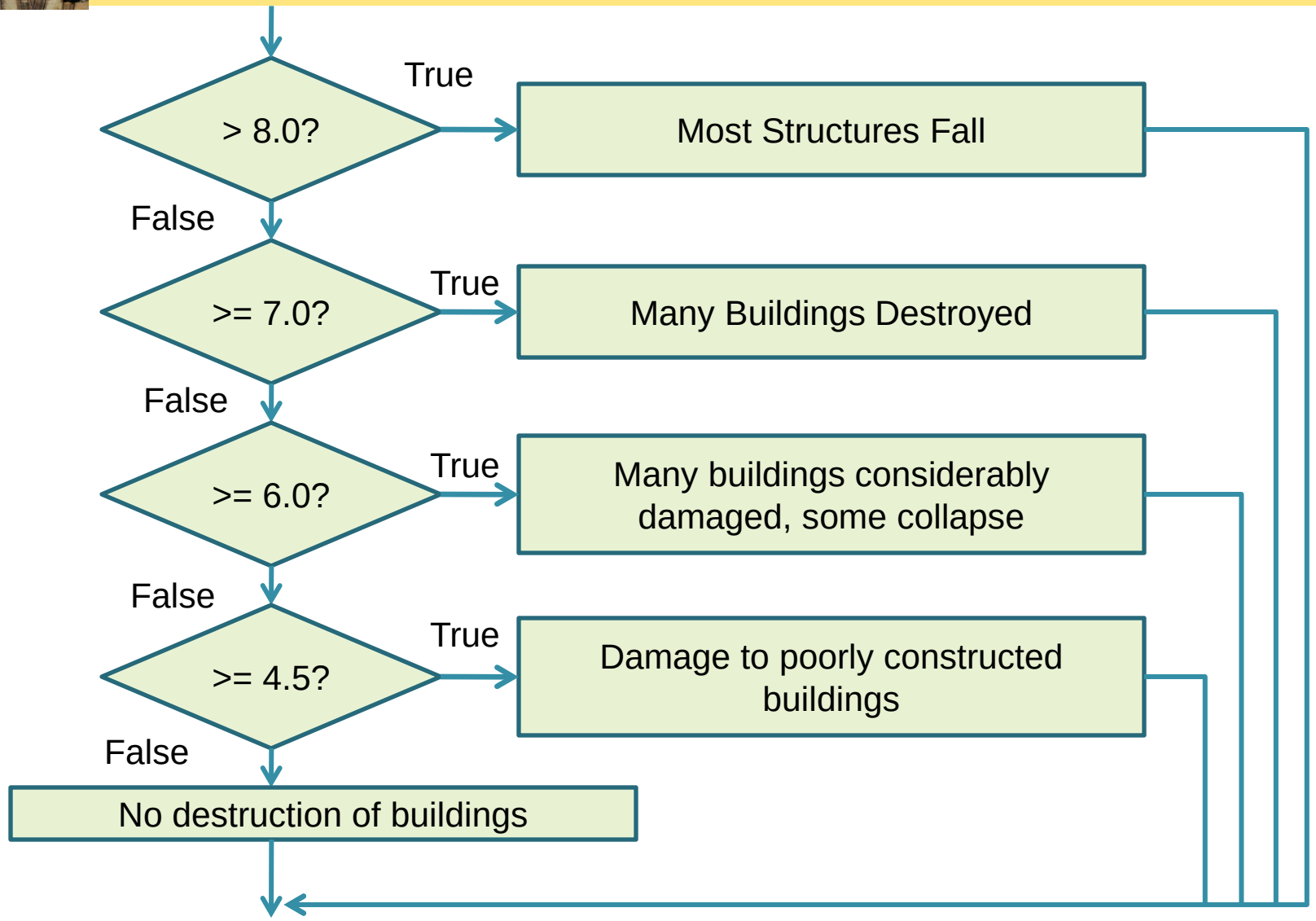
When using multiple `if` statements, test general conditions after more specific conditions.

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

Copyright © 2013 by John Wiley & Sons. All rights reserved.

Flowchart of Multiway branching



Copyright © 2013 by John Wiley & Sons. All rights reserved.

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");
}
```

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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");
}
```

If the earthquake scale is 8.0, all statements will be executed.

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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.

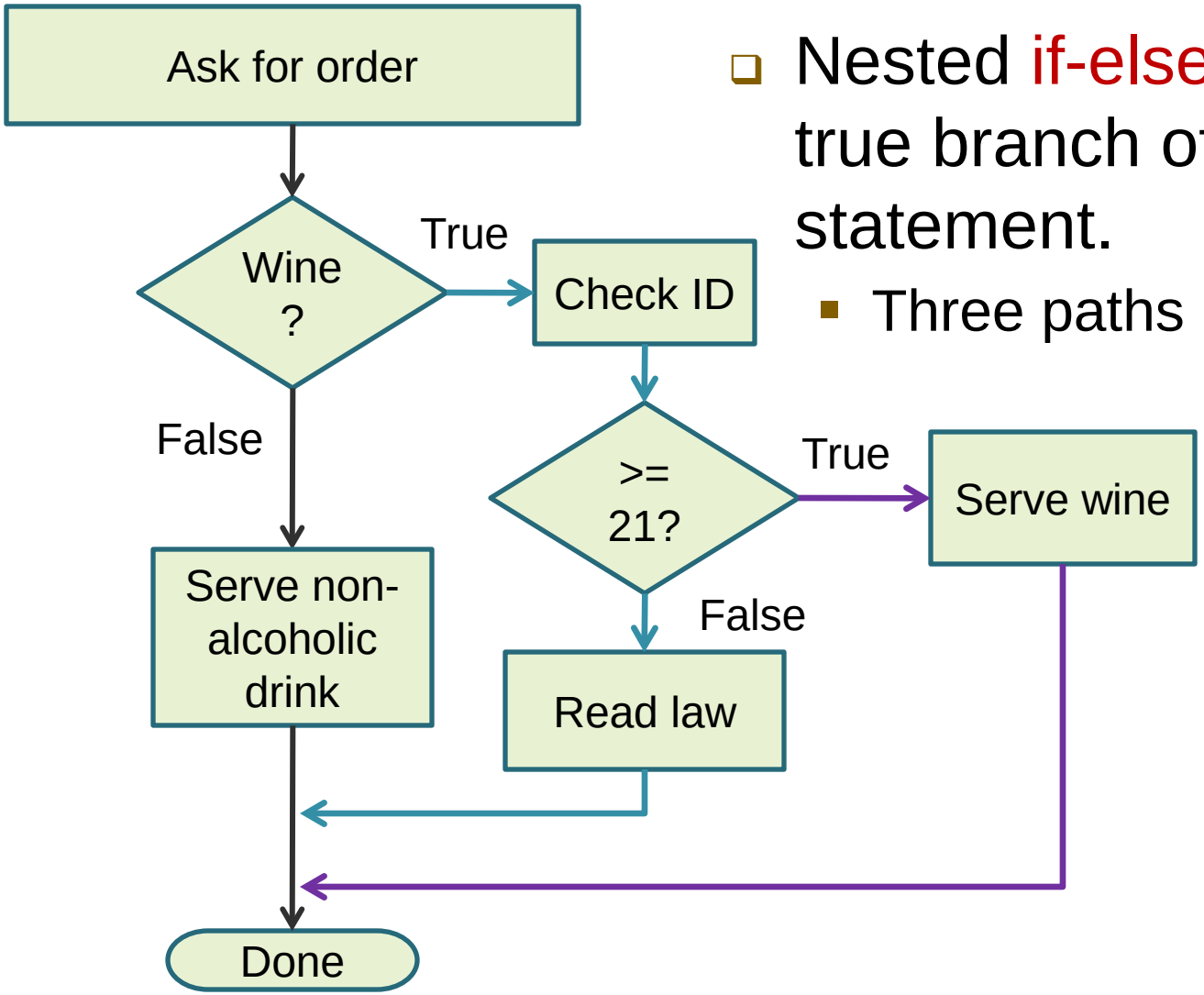
Copyright © 2013 by John Wiley & Sons. All rights reserved.

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

Copyright © 2013 by John Wiley & Sons. All rights reserved.

Flowchart of a Nested if



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

Copyright © 2013 by John Wiley & Sons. All rights reserved.

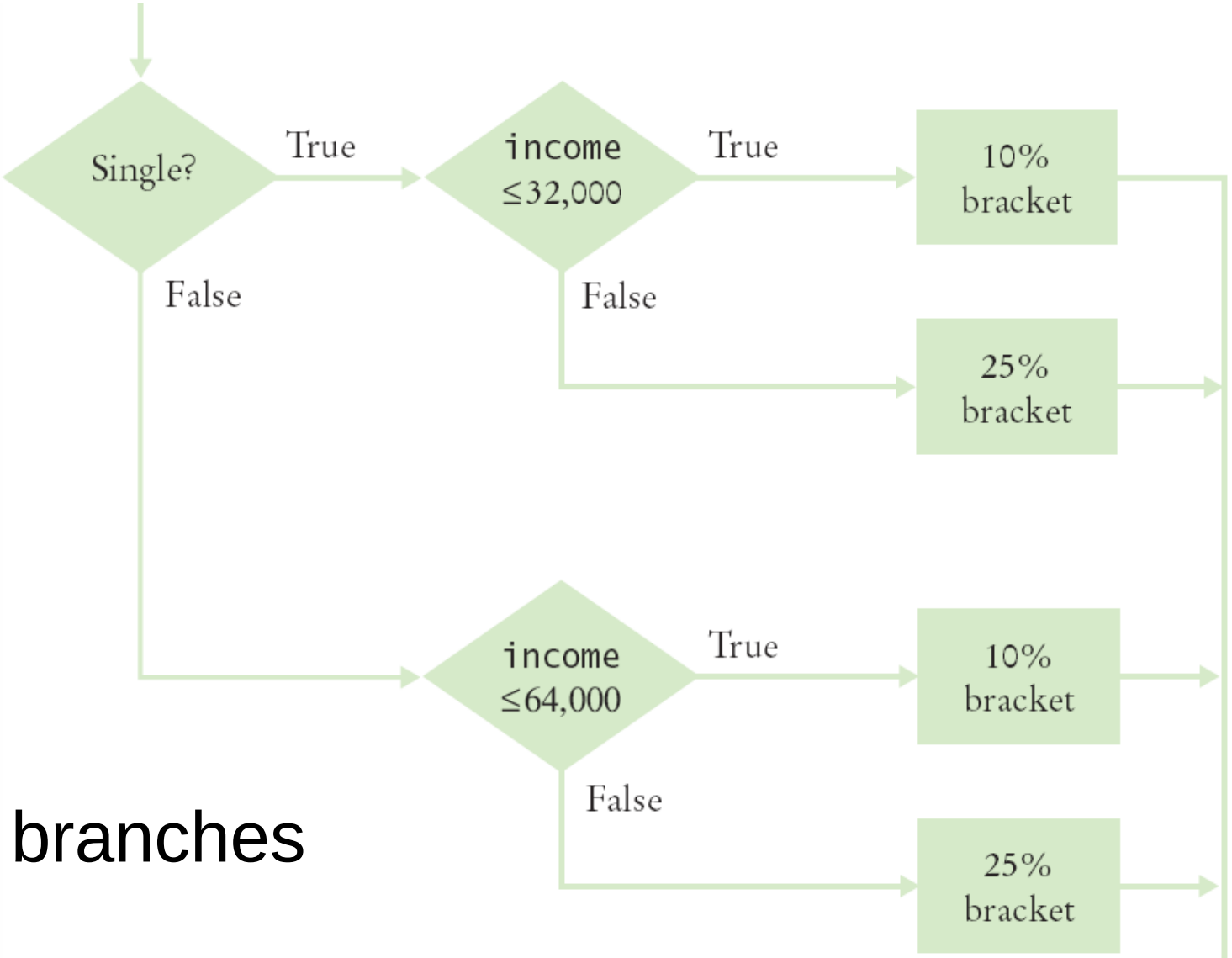
Tax Example: Nested `if`s

- 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

Copyright © 2013 by John Wiley & Sons. All rights reserved.

Flowchart for Tax Example



- Four branches

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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();
26
```

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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      }
```

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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
```

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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
```

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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    {
```

Copyright © 2013 by John Wiley & Sons. All rights reserved.

Hand-Tracing Tax Example (3)

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

tax1	tax2	income	marital status
~~0~~	~~0~~	80000	m
6400	4000		

```
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       }
```

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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  }
```

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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`**

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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) . . .
```

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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:

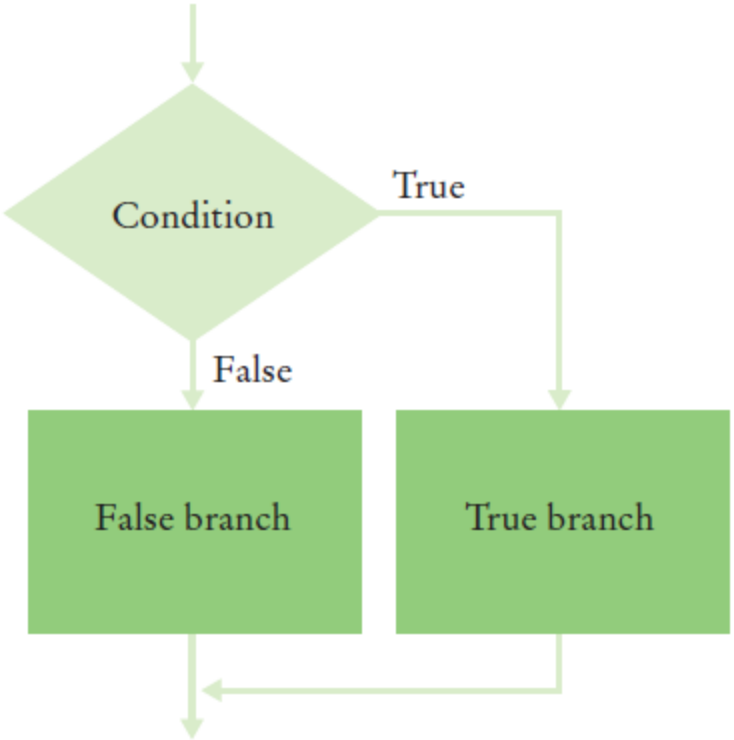
- Connect them with arrows
  - But never point an arrow inside another branch!

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

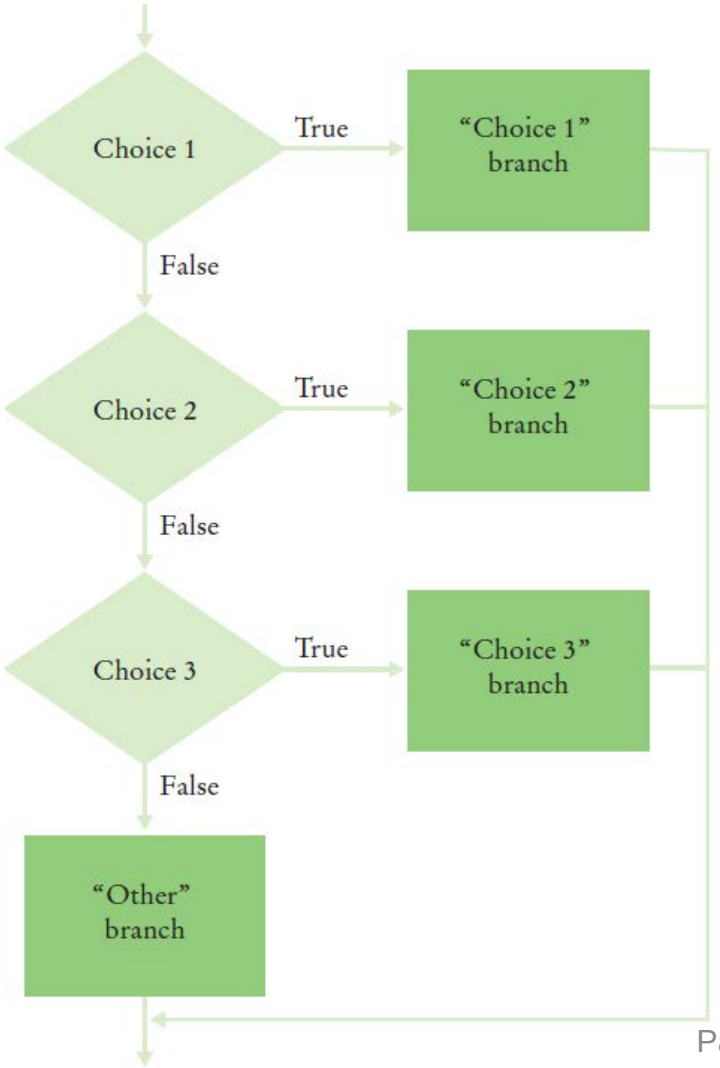
Copyright © 2013 by John Wiley & Sons. All rights reserved.

Conditional Flowcharts

- Two Outcomes



- Multiple Outcomes

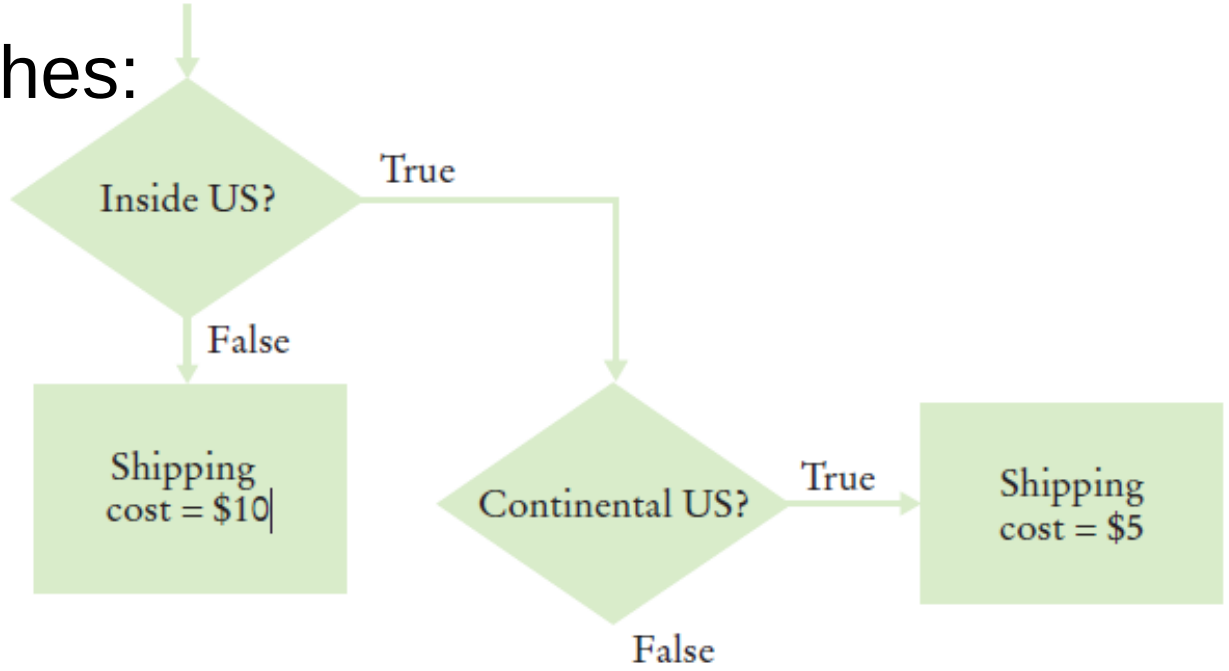


Copyright © 2013 by John Wiley & Sons. All rights reserved.

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:

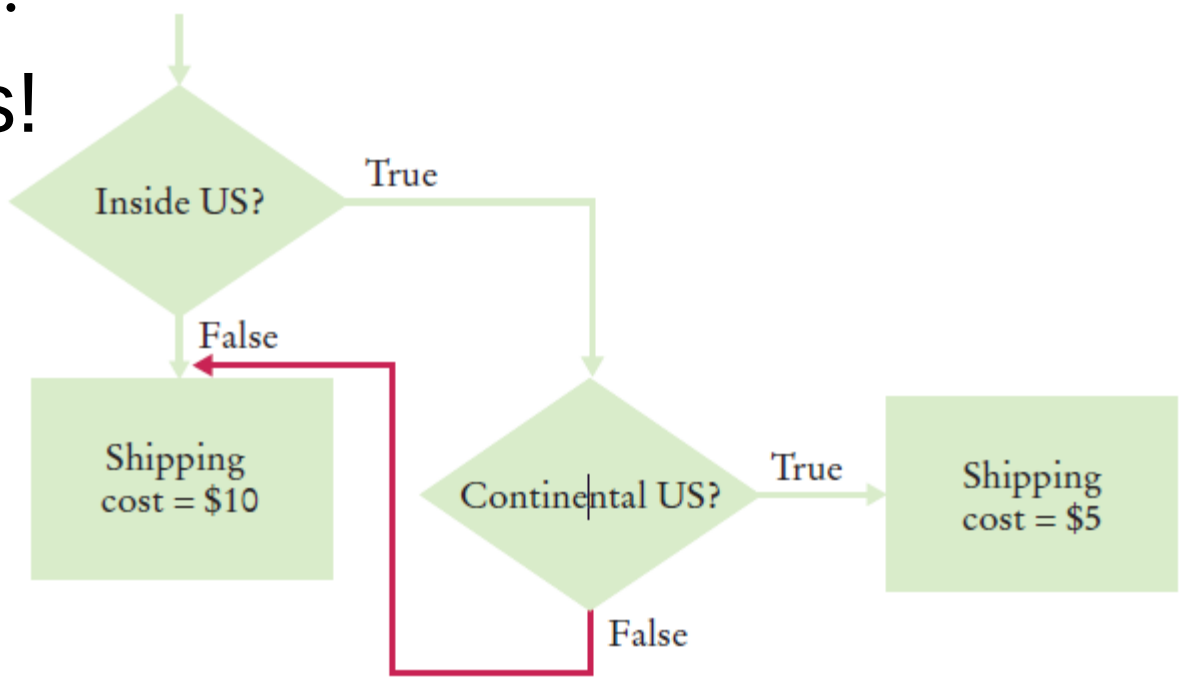


Copyright © 2013 by John Wiley & Sons. All rights reserved.

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!

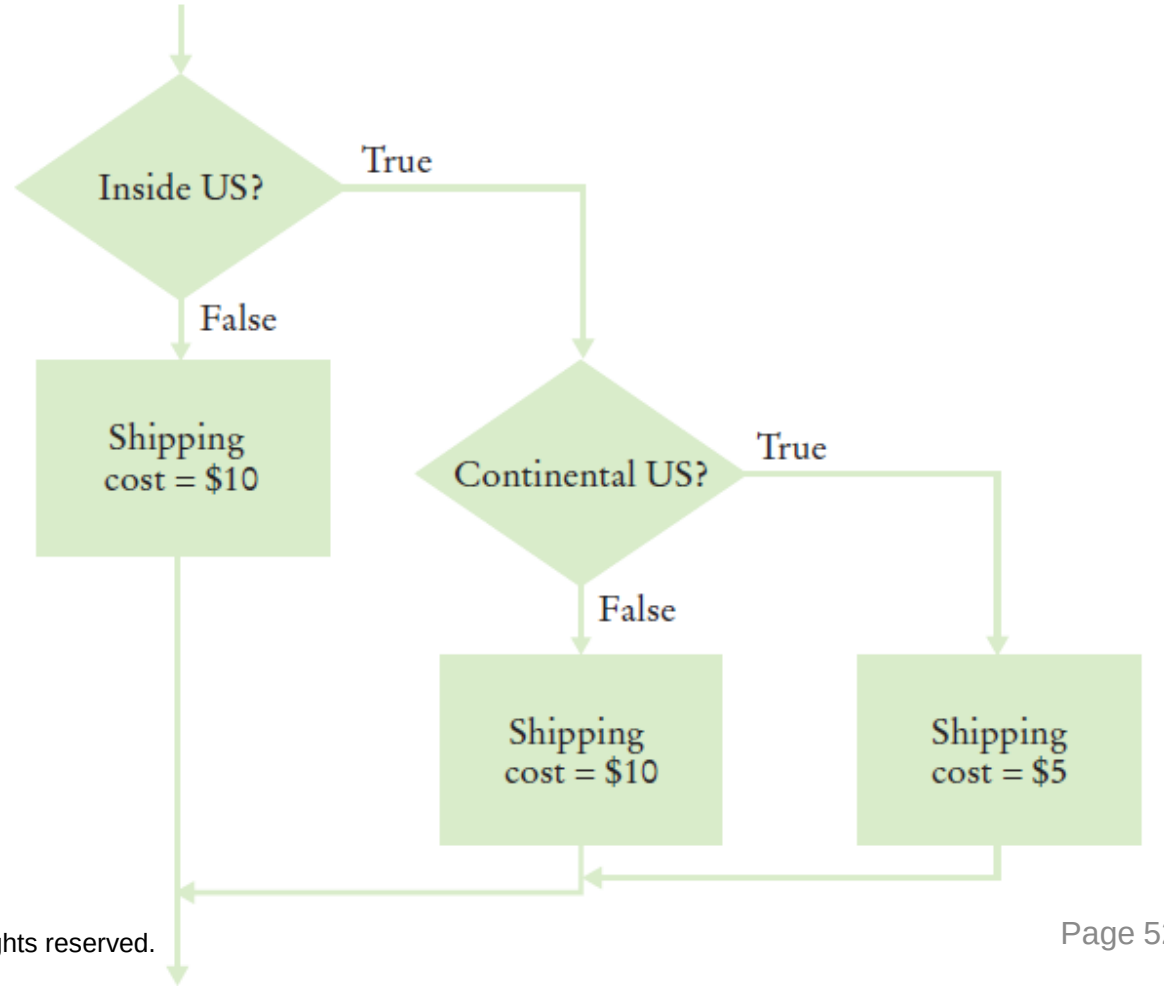


Copyright © 2013 by John Wiley & Sons. All rights reserved.

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:



Copyright © 2013 by John Wiley & Sons. All rights reserved.

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

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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% of 40,000
50,000 m	5,000	10% bracket
104,000 m	16,400	6,400 + 25% of 40,000
32,000 m	3,200	boundary case
0	0	boundary case

Copyright © 2013 by John Wiley & Sons. All rights reserved.

Code Coverage

- **Black-box testing:** Test functionality without consideration of internal structure of implementation
- **White-box testing:** Take internal structure into account when designing tests
- **Test coverage:** Measure of how many parts of a program have been tested
- Make sure that each part of your program is exercised at least once by one test case
  E.g., make sure to execute each branch in at least one test case

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Code Coverage

- Include boundary test cases: Legal values that lie at the boundary of the set of acceptable inputs
- Tip: Write first test cases before program is written completely → gives insight into what program should do

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

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

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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");
}
```

Truth table

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

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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");
}
```

A	!A
true	false
false	true

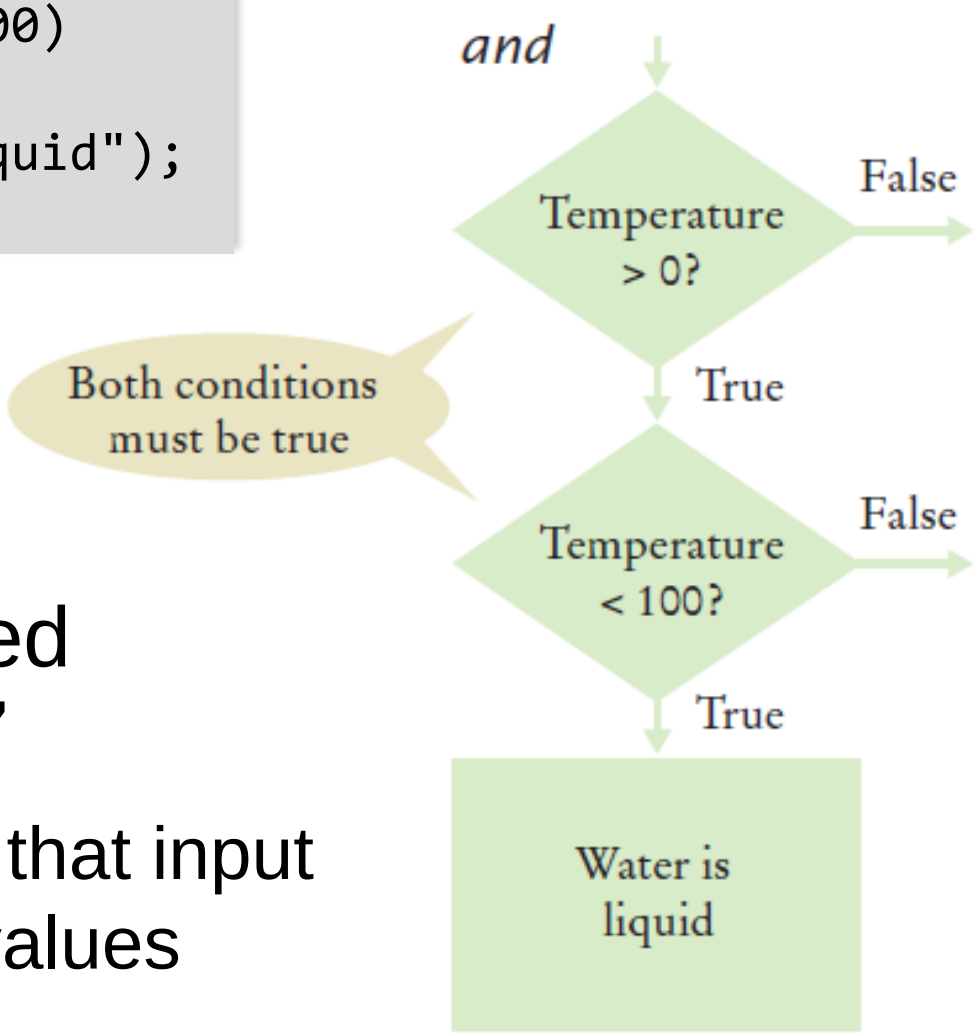
- If using !, try to use simpler logic:

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

Copyright © 2013 by John Wiley & Sons. All rights reserved.

and Flowchart

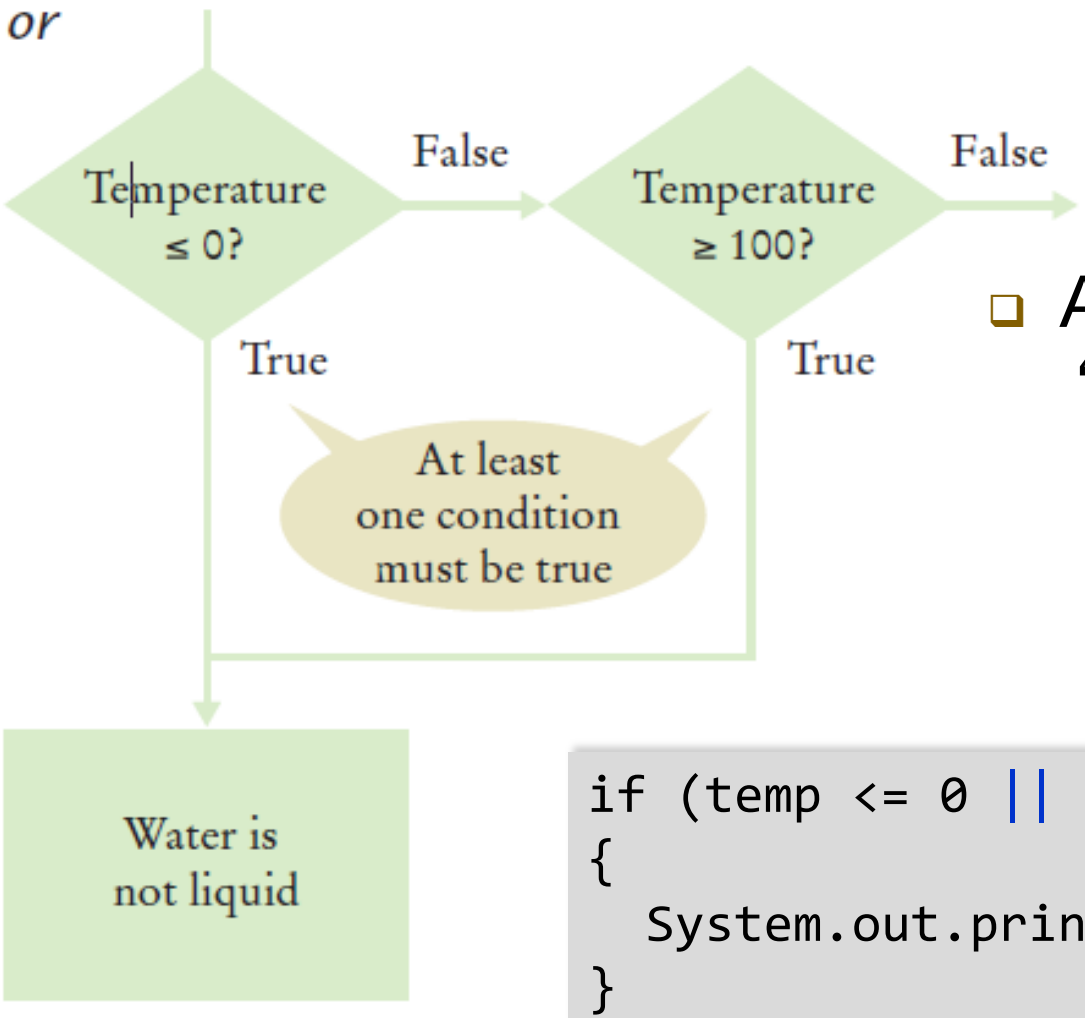
```
if (temp > 0 && temp < 100)
{
   System.out.println("Liquid");
}
```



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

Copyright © 2013 by John Wiley & Sons. All rights reserved.

or Flowchart



- Another form of 'range checking'
  - Checks if value is **outside** a range

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

Copyright © 2013 by John Wiley & Sons. All rights reserved.

Boolean Operator Examples

Table 5 Boolean Operator Examples

Expression	Value	Comment
`0 < 200 && 200 < 100`	`false`	Only the first condition is true.
`0 < 200 || 200 < 100`	`true`	The first condition is true.
`0 < 200 || 100 < 200`	`true`	The `||` is not a test for "either-or". If both conditions are true, the result is true.
`0 < x && x < 100 || x == -1`	`(0 < x && x < 100) || x == -1`	The `&&` operator has a higher precedence than the `||` operator (see Appendix B).
		

Copyright © 2013 by John Wiley & Sons. All rights reserved.

Boolean Operator Examples

Table 5 Boolean Operator Examples

Copyright © 2013 by John Wiley & Sons. All rights reserved.

Using Boolean Variables

- `private boolean married;`
- Set to truth value:

```
married = input.equals("M");
```

- Use in conditions:

```
if (married) ... else ...
if (!married) ...
```

- Also called *flag*
- It is considered gauche to write a test such as

```
if (married == true) ... // Don't
```

- Just use the simpler test

```
if (married) ...
```

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

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)
```

Copyright © 2013 by John Wiley & Sons. All rights reserved.

Common Error 3.6

Confusing && and || Conditions

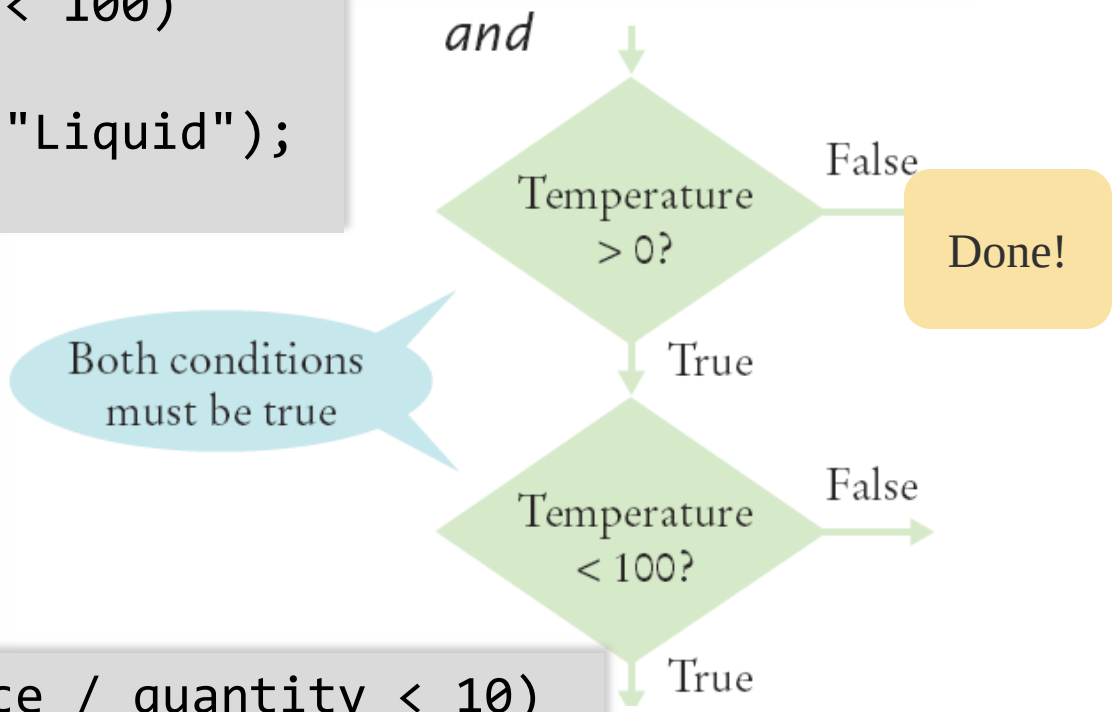
- It is a surprisingly common error to confuse && and || conditions.
- A value lies between 0 and 100 if it is at least 0 ***and*** at most 100.
- It lies outside that range if it is less than 0 ***or*** greater than 100.
- There is no golden rule; you just have to think carefully.

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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)
```

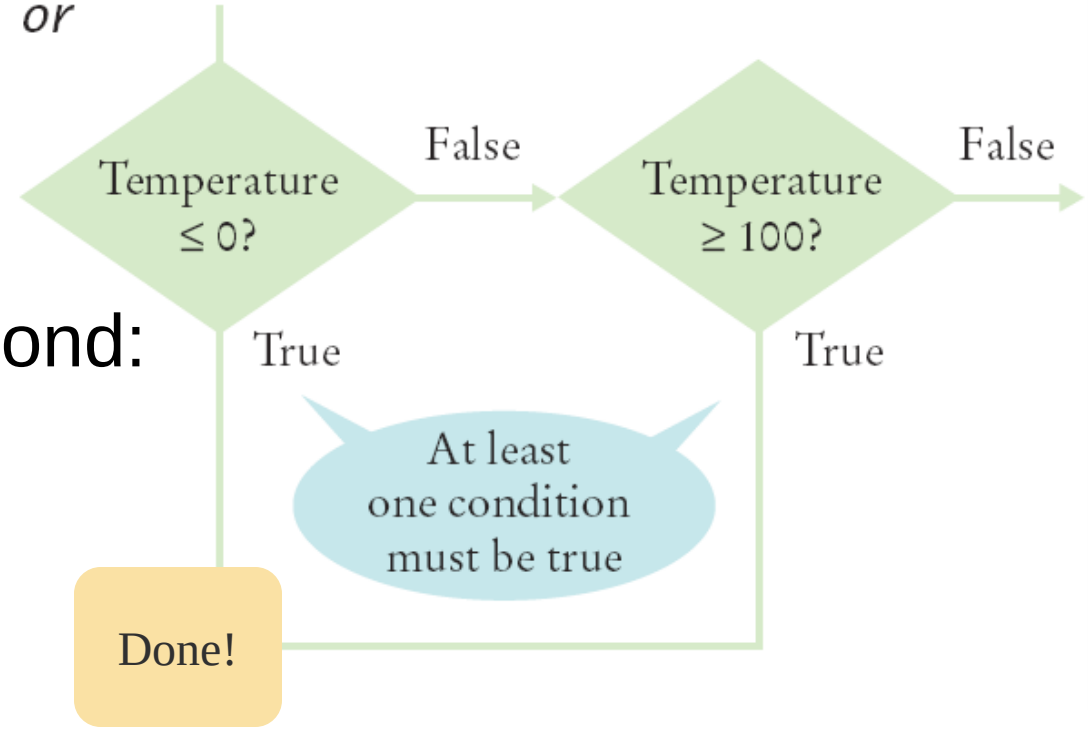
Copyright © 2013 by John Wiley & Sons. All rights reserved.

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");
}
```

or



- Java doesn't!
- Don't do these second:
  - Assignment
  - Output

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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.

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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)
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

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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.
```

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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.

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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.

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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.

Copyright © 2011 by John Wiley & Sons. All rights reserved.