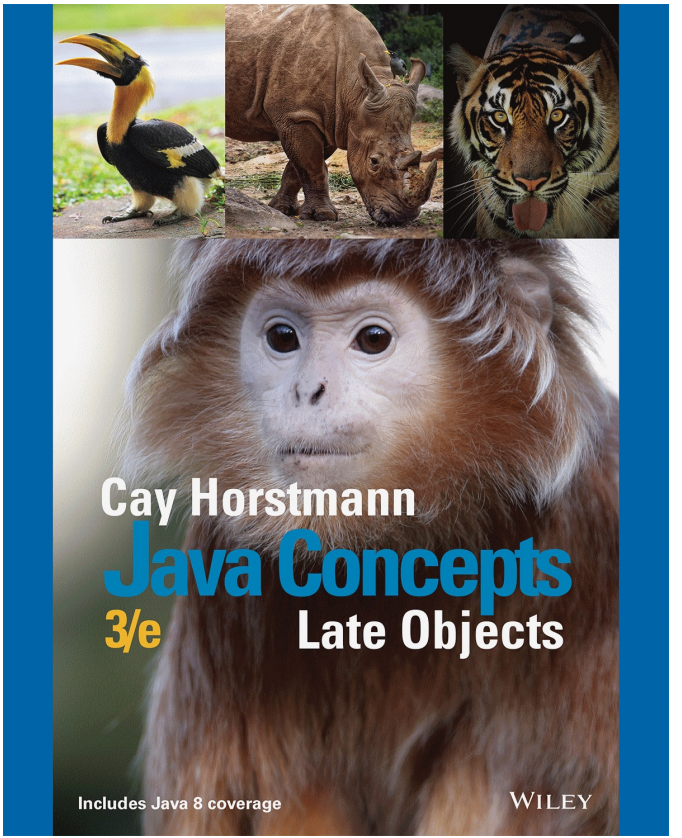
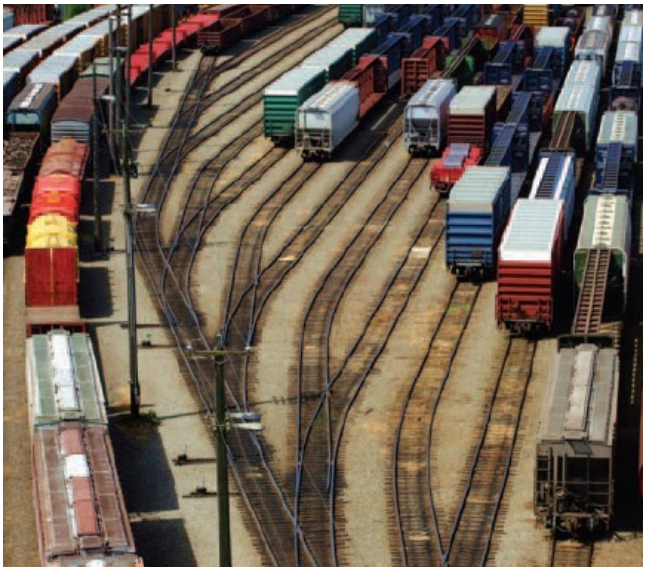
Chapter 3 - Decisions



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 for validate user input

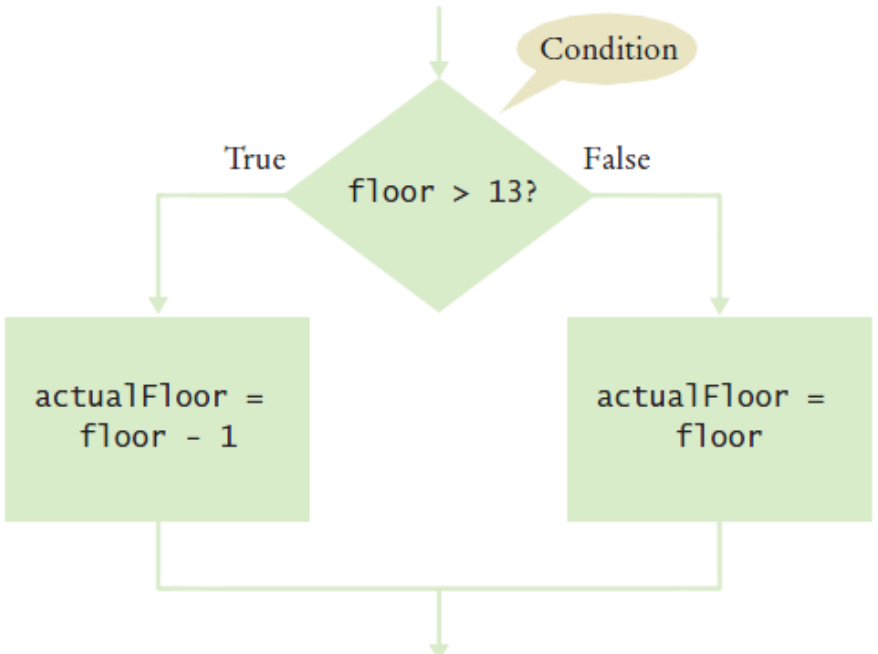
The `if` Statement

- 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.

Flowchart of the `if` Statement

- One of the two branches is executed once

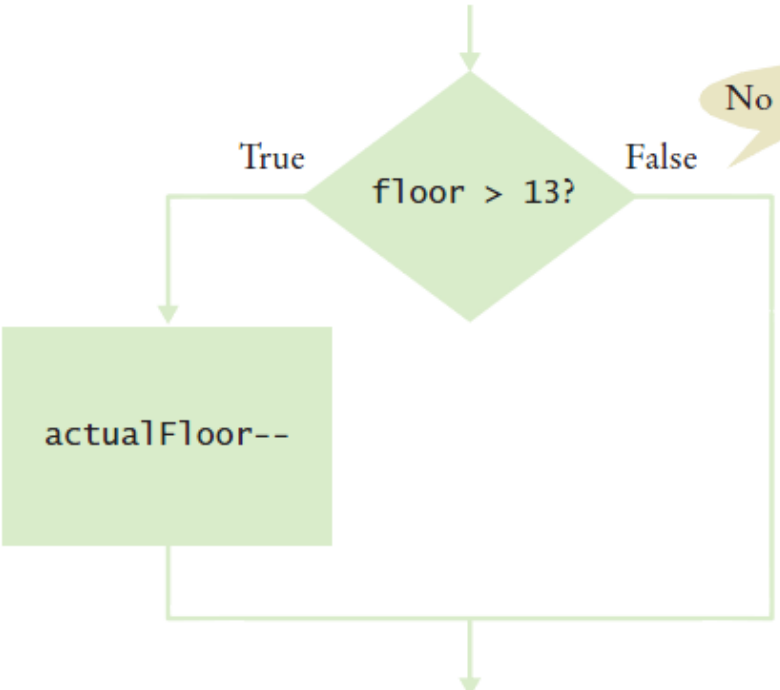
  True (`if`) branch or False (`else`) branch




```
int actualFloor;

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

Flowchart with only a True Branch

- An `if` statement may not need a 'False' (`else`) branch



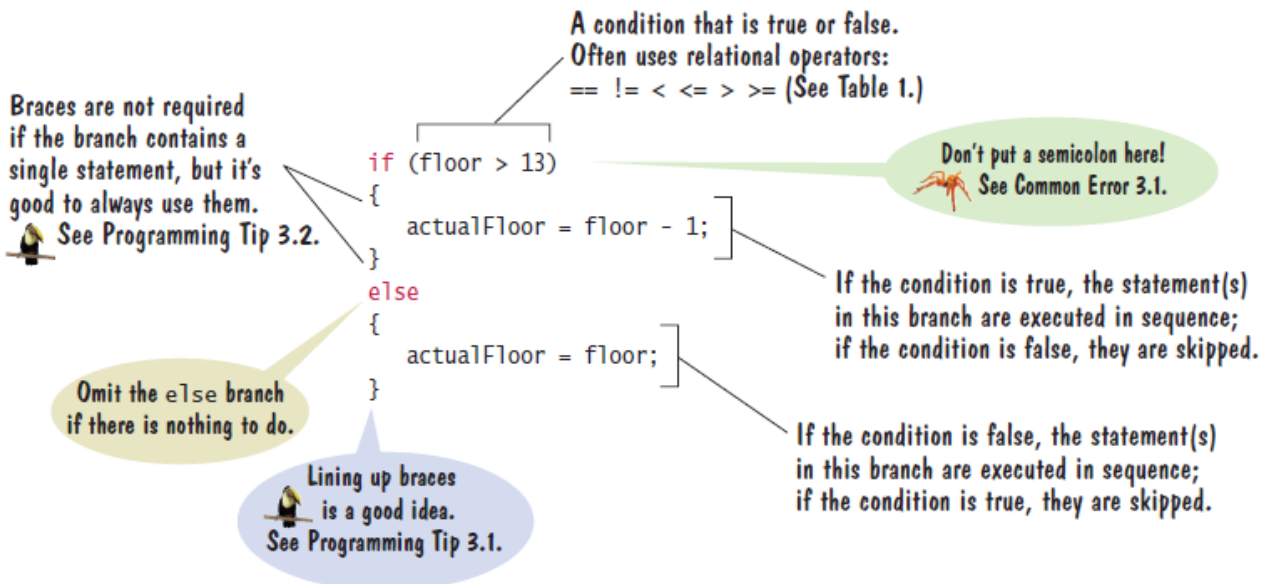

```
int actualFloor = floor;

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

Syntax 3.1 The `if` Statement

Syntax

```
if (condition)
{
   statements
}
```

```
if (condition) { statements1 }
else { statements2 }
```



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
```

Self Check 3.1

In some Asian countries, the number 14 is considered unlucky. Some building owners play it safe and *skip* both the thirteenth and the fourteenth floor. How would you modify the sample program to handle such a building?

Answer: Change the `if` statement to

```
if (floor > 14)
{
   actualFloor = floor - 2;
}
```

Self Check 3.2

Consider the following if statement to compute a discounted price:

```
if (originalPrice > 100)
{
   discountedPrice = originalPrice - 20;
}
else
{
   discountedPrice = originalPrice - 10;
}
```

What is the discounted price if the original price is 95? 100? 105?

Answer: 85. 90. 85.

Self Check 3.3

Compare this if statement with the one in Self Check 2:

```
if (originalPrice < 100)
{
   discountedPrice = originalPrice - 10;
}
else
{
   discountedPrice = originalPrice - 20;
}
```

Do the two statements always compute the same value? If not, when do the values differ?

Answer: The only difference is if `originalPrice` is 100. The statement in Self Check 2 sets `discountedPrice` to 90; this one sets it to 80.

Self Check 3.4

Consider the following statements to compute a discounted price:

```
discountedPrice = originalPrice;
if (originalPrice > 100)
{
   discountedPrice = originalPrice - 10;
}
```

What is the discounted price if the original price is 95? 100? 105?

Answer: 95. 100. 95.

Self Check 3.5

The variables `fuelAmount` and `fuelCapacity` hold the actual amount of fuel and the size of the fuel tank of a vehicle. If less than 10 percent is remaining in the tank, a status light should show a red color; otherwise it shows a green color. Simulate this process by printing out either "`red`" or "`green`".

Answer:

```
if (fuelAmount < 0.10 * fuelCapacity)
{
   System.out.println("red");
}
else
{
   System.out.println("green");
}
```

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--;
}
```

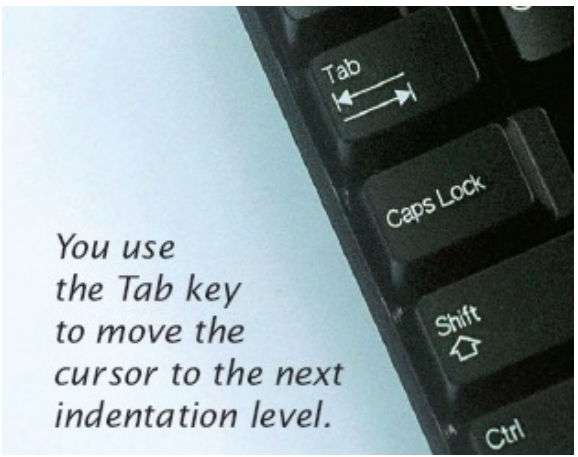
```
if (floor > 13)
   floor--;
```

- 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

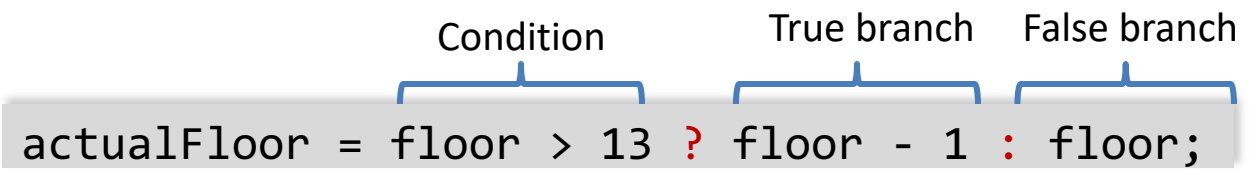
- A semicolon after an `if` statement
- It is easy to forget and add a semicolon after an `if` statement
  - The true path is now the space just before the semicolon

```
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



  - 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

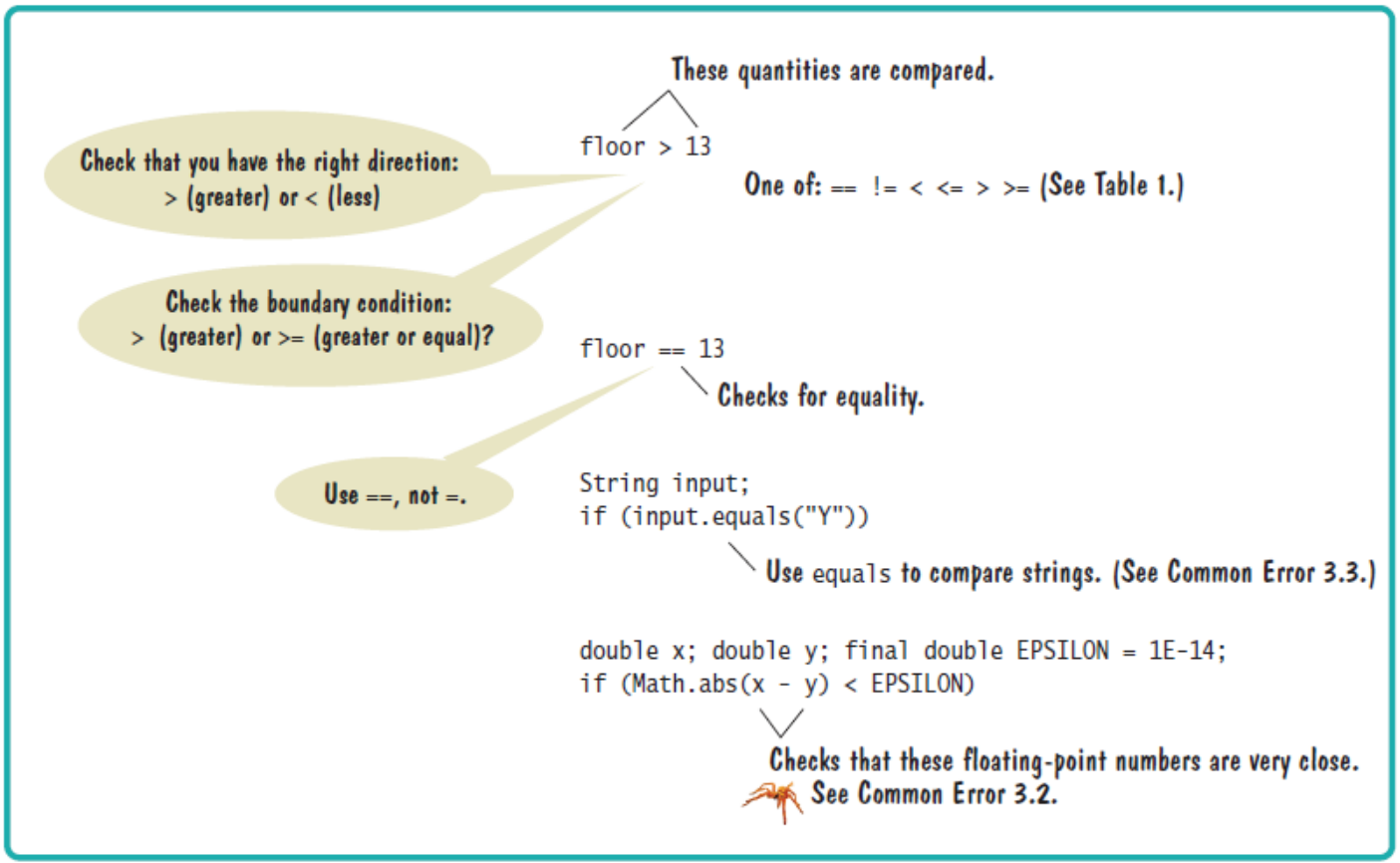
```
if (floor > 13)
   ..
if (floor >= 13)
   ..
if (floor < 13)
   ..
if (floor <= 13)
   ..
if (floor == 13)
   ..
```

Beware!

Table 1 Relational Operators

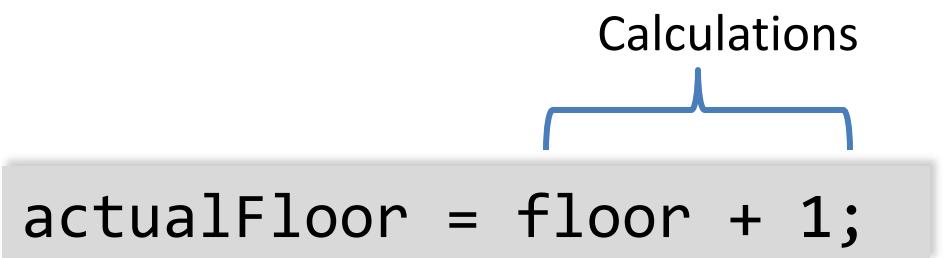
Java	Math Notation	Description
>	>	Greater than
>=	≥	Greater than or equal
<	<	Less than
<=	≤	Less than or equal
==	=	Equal
!=	≠	Not equal

Syntax 3.2 Comparisons



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




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

Relational Operator Use

Table 2 Relational Operator Examples

Expression	Value	Comment
`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.
`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 93.
`"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 94.

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)) ...
```

Self Check 3.6

Which of the following conditions are true, provided `a` is 3 and `b` is 4?

a. `a + 1 <= b`

b. `a + 1 >= b`

c. `a + 1 != b`

Answer: (a) and (b) are both true, (c) is false.

Self Check 3.7

Give the opposite of the condition

```
floor > 13
```

Answer: `floor <= 13`

Self Check 3.8

What is the error in this statement?

```
if (scoreA = scoreB)
{
   System.out.println("Tie");
}
```

Answer: The values should be compared with ==, not =.

Self Check 3.9

Supply a condition in this `if` statement to test whether the user entered a Y:

```
System.out.println("Enter Y to quit.");
String input = in.next();
if (. . .)
{
   System.out.println("Goodbye.");
}
```

Answer: `input.equals("Y")`

Self Check 3.10

How do you test that a string `str` is the empty string?

Answer: `str.equals("")` or `str.length() == 0`

Common Error

- 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

- 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)

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

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;
```

Multiple Alternatives

- What if you have more than two branches?
- Count the branches for the following earthquake effect example:

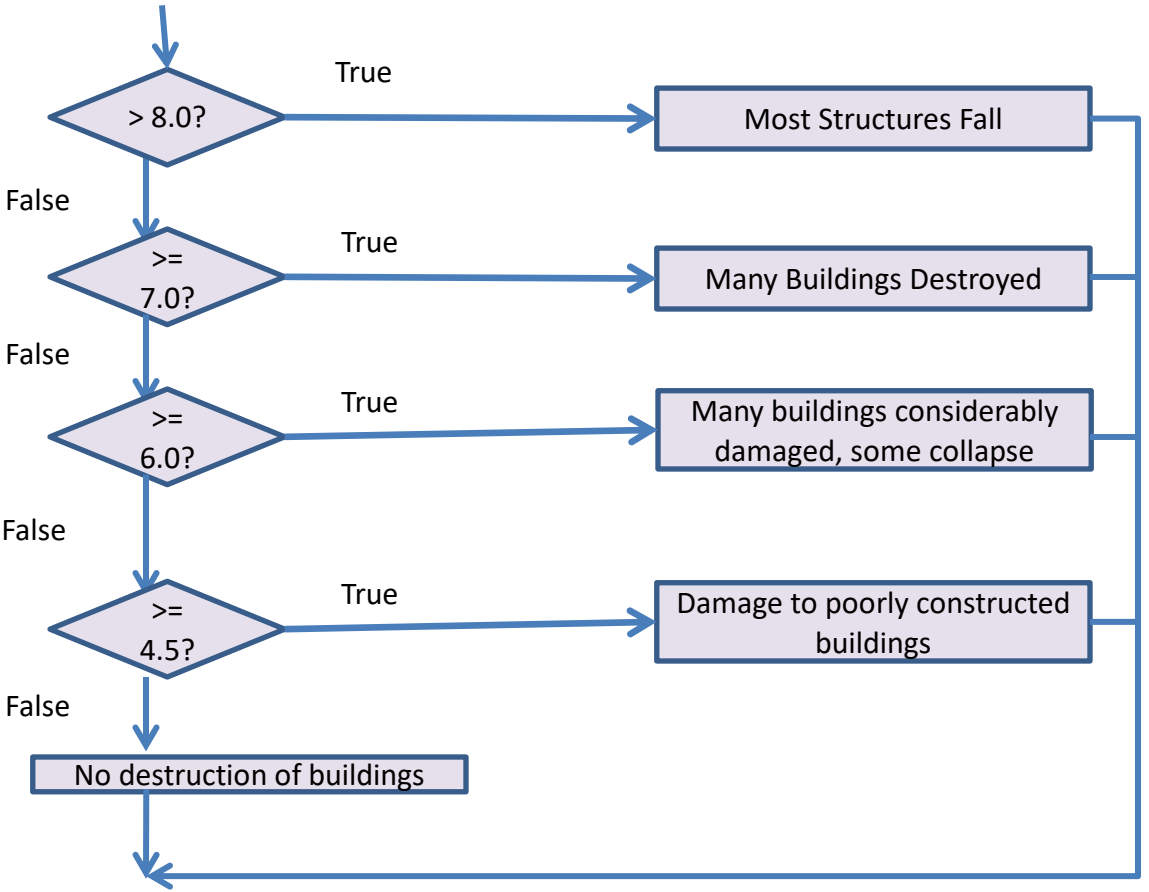
- 8.0 (or greater)
- >= 7.0 but < 8.0
- >= 6.0 but < 7.0
- >= 4.5 but < 6.00
- Less than 4.5

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.

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");
}
```

Self Check 3.11

In a game program, the scores of players A and B are stored in variables `scoreA` and `scoreB`. Assuming that the player with the larger score wins, write an `if`/`else if`/`else` sequence that prints out "`A won`", "`B won`", or "`Game tied`".

Answer:

```
if (scoreA > scoreB)
{
   System.out.println("A won");
}
else if (scoreA < scoreB)
{
   System.out.println("B won");
}
else
{
   System.out.println("Game tied");
}
```

Self Check 3.12

Write a conditional statement with three branches that sets s to 1 if x is positive, to –1 if x is negative, and to 0 if x is zero.

Answer:

```
if (x > 0) { s = 1; }
else if (x < 0) { s = -1; }
else { s = 0; }
```

Self Check 3.13

How could you achieve the task of Self Check 12 with only two branches?

Answer: You could first set s to one of the three values:

```
s = 0;
if (x > 0) { s = 1; }
else if (x < 0) { s = -1; }
```

Self Check 3.14

Beginners sometimes write statements such as the following:

```
if (price > 100)
{
   discountedPrice = price - 20;
}
else if (price <= 100)
{
   discountedPrice = price - 10;
}
```

Explain how this code can be improved.

Answer: The `if (price <= 100)` can be omitted (leaving just `else`), making it clear that the `else` branch is the sole alternative.

Self Check 3.15

Suppose the user enters -1 into the earthquake program. What is printed?

Answer: No destruction of buildings.

Self Check 3.16

Suppose we want to have the earthquake program check whether the user entered a negative number. What branch would you add to the `if` statement, and where?

Answer: Add a branch before the final else:

```
else if (richter < 0)
{
   System.out.println("Error: Negative input");
}
```

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.

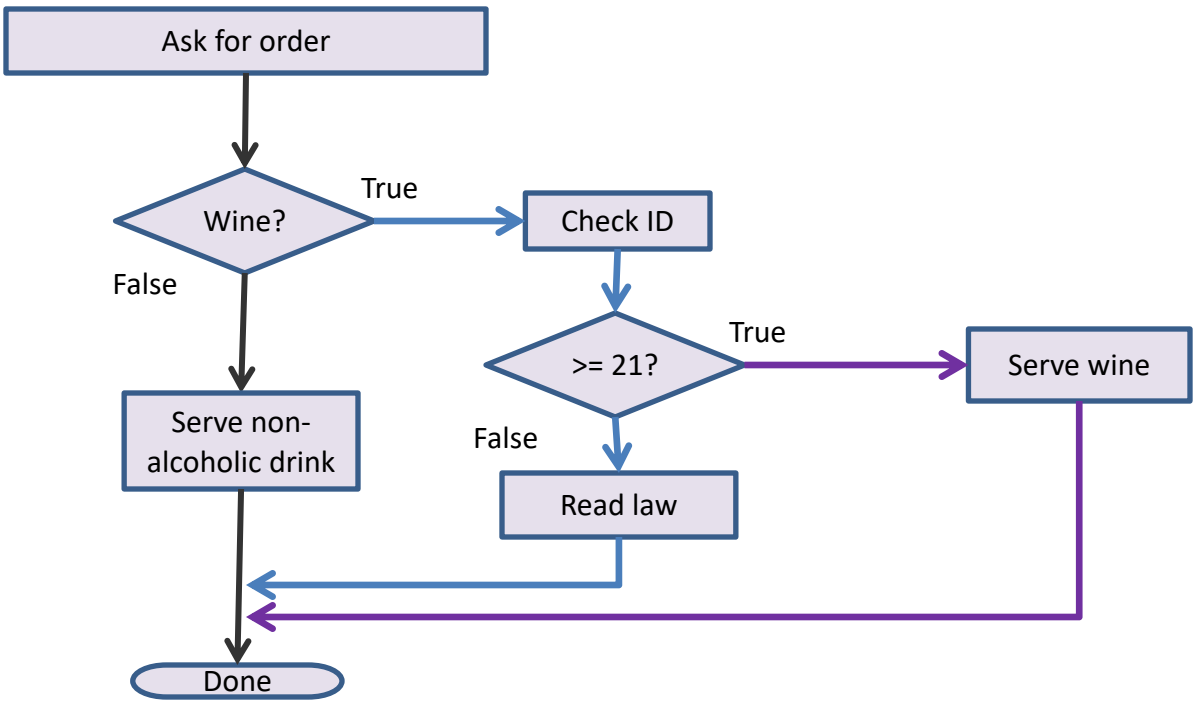
```
int digit = . . .;
switch (digit)
{
   case 1: digitName = "one";   break;
   case 2: digitName = "two";   break;
   case 3: digitName = "three"; break;
   case 4: digitName = "four";  break;
   case 5: digitName = "five";  break;
   case 6: digitName = "six";   break;
   case 7: digitName = "seven";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

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



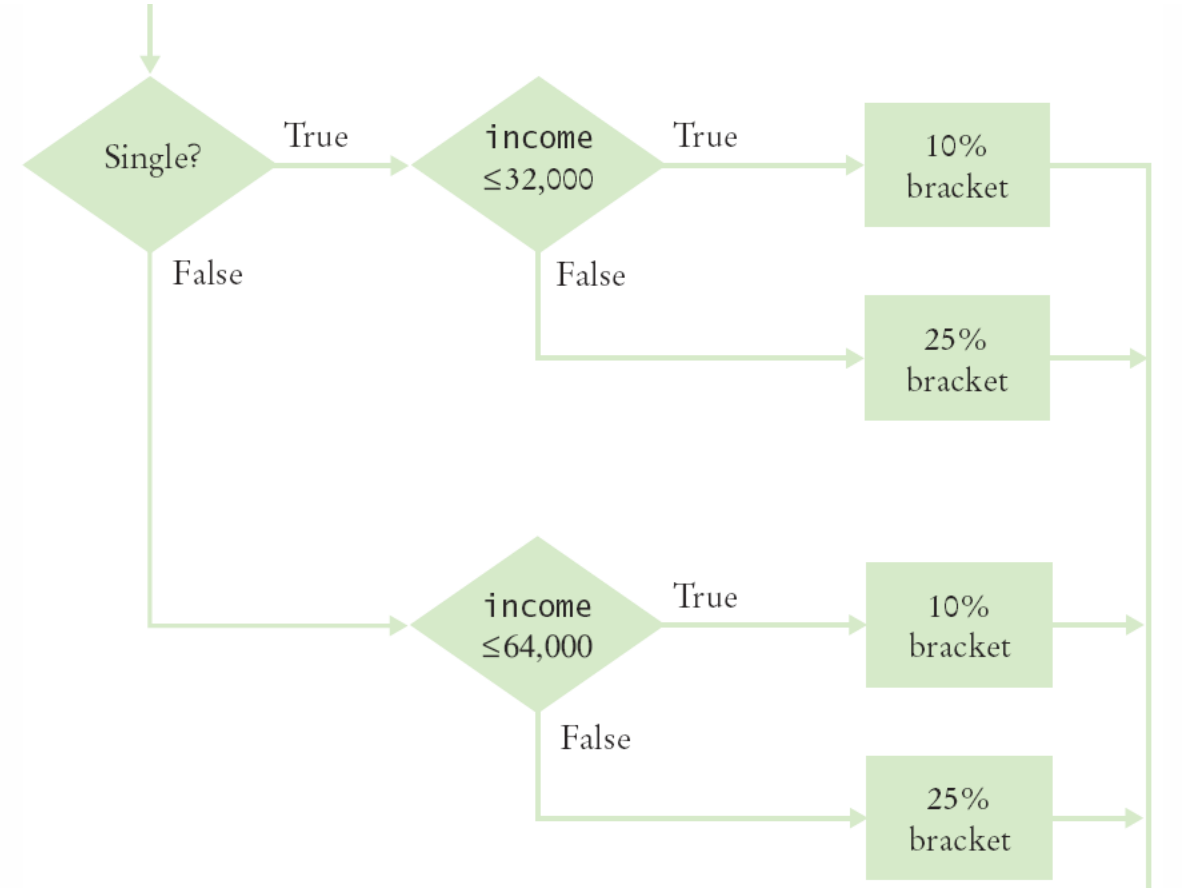
Tax Example: Nested `if`s

- 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

Flowchart for Tax Example



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
```

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
```

Self Check 3.17

What is the amount of tax that a single taxpayer pays on an income of $32,000?

Answer: 3200.

Self Check 3.18

Would that amount change if the first nested `if` statement changed from

```
if (income <= RATE1_SINGLE_LIMIT)
```

to

```
if (income < RATE1_SINGLE_LIMIT)
```

Answer: No. Then the computation is 0.10 × 32000 + 0.25 × (32000 – 32000).

Self Check 3.19

Suppose Harry and Sally each make $40,000 per year. Would they save taxes if they married?

Answer: No. Their individual tax is $5,200 each, and if they married, they would pay $10,400. Actually, taxpayers in higher tax brackets (which our program does not model) may pay higher taxes when they marry, a phenomenon known as the *marriage penalty.*

Self Check 3.20

How would you modify the `TaxCalculator.java` program in order to check that the user entered a correct value for the marital status (i.e., `s` or `m`)?

Answer: Change `else` in line 41 to

```
else if (maritalStatus.equals("m"))
```

and add another branch after line 52:

```
else
{
   System.out.println(
      "Error: marital status should be s or m.");
}
```

Self Check 3.21

Some people object to higher tax rates for higher incomes, claiming that you might end up with less money after taxes when you get a raise for working hard. What is the flaw in this argument?

Answer: The higher tax rate is only applied on the income in the higher bracket. Suppose you are single and make $31,900. Should you try to get a $200 raise? Absolutely: you get to keep 90 percent of the first $100 and 75 percent of the next $100.

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)

- Setup
  - Table of variables
  - Initial values

tax1	tax2	income	marital status
0	0		

```
 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)

- Input variables
  - From user
  - Update table

tax1	tax2	income	marital status
0	0	80000	m

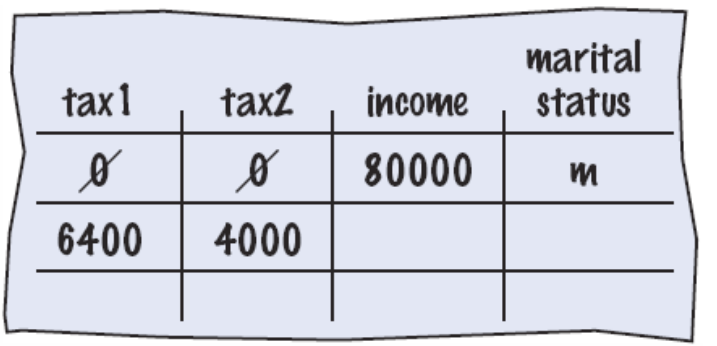
```
20    Scanner in = new Scanner(System.in);
21    System.out.print("Please enter your income: ");
22    double income = in.nextDouble();
23
24    System.out.print("Please enter s for single, m for married: ");
25    String maritalStatus = in.next();
```

- 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 <= 64000, we move to the else clause on line 47
  - Update variables on lines 49 and 50
  - Use constants

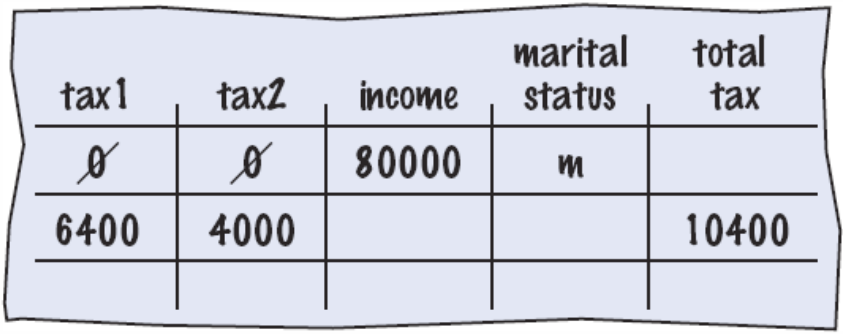


tax1	tax2	income	marital status
~~0~~	~~0~~	80000	m
6400	4000		

```
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       }
```

Hand-Tracing Tax Example (4)

- Output
  - Calculate
  - As expected?



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

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

Common Error

- 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:

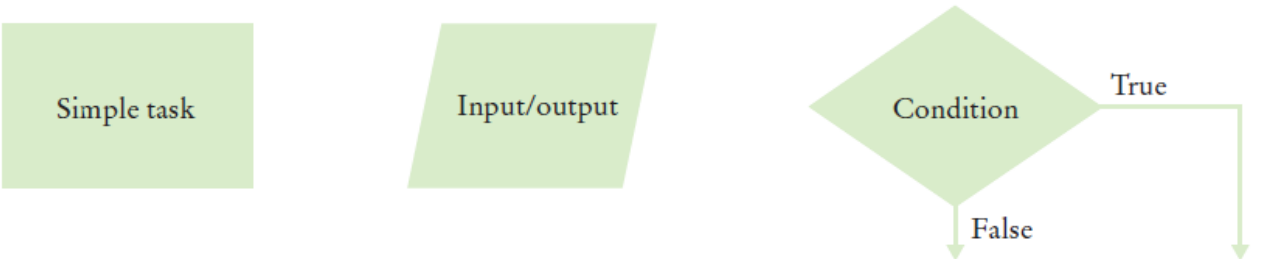
```
FilingStatus status = FilingStatus.SINGLE;
```

  - And you can use the comparison operator with them:

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

Problem Solving: Flowcharts

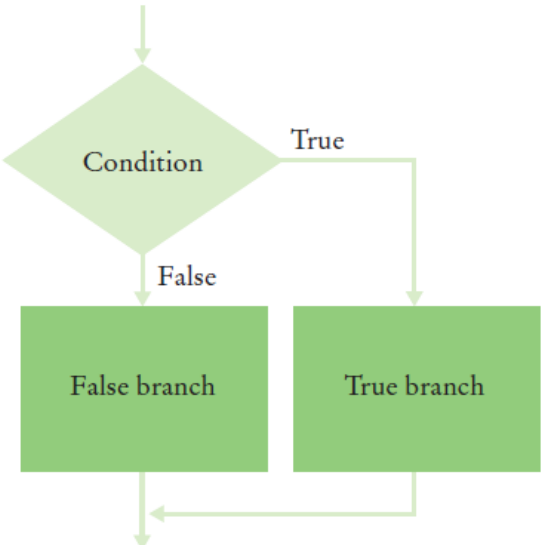
- You have seen a few basic flowcharts
- A flowchart shows the structure of decisions and tasks to solve a problem
- Basic flowchart elements:



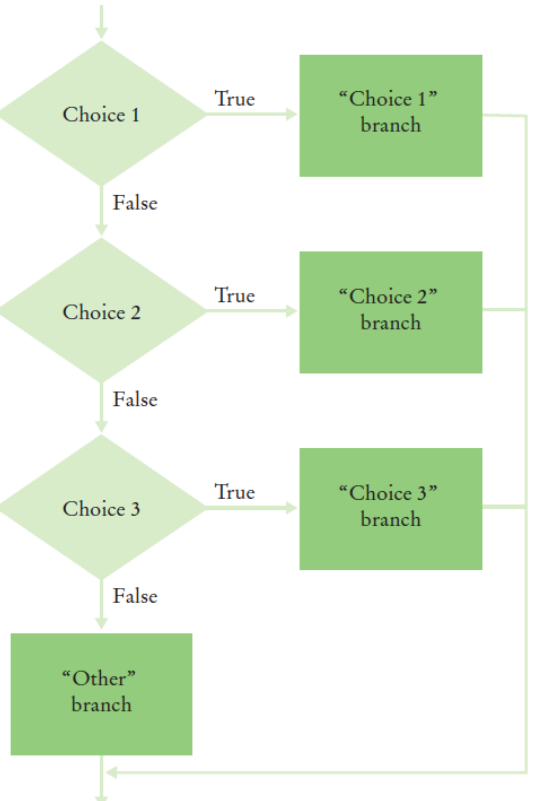
- Connect them with arrows
  - But never point an arrow
  - inside another branch!

Conditional Flowcharts

- Two Outcomes



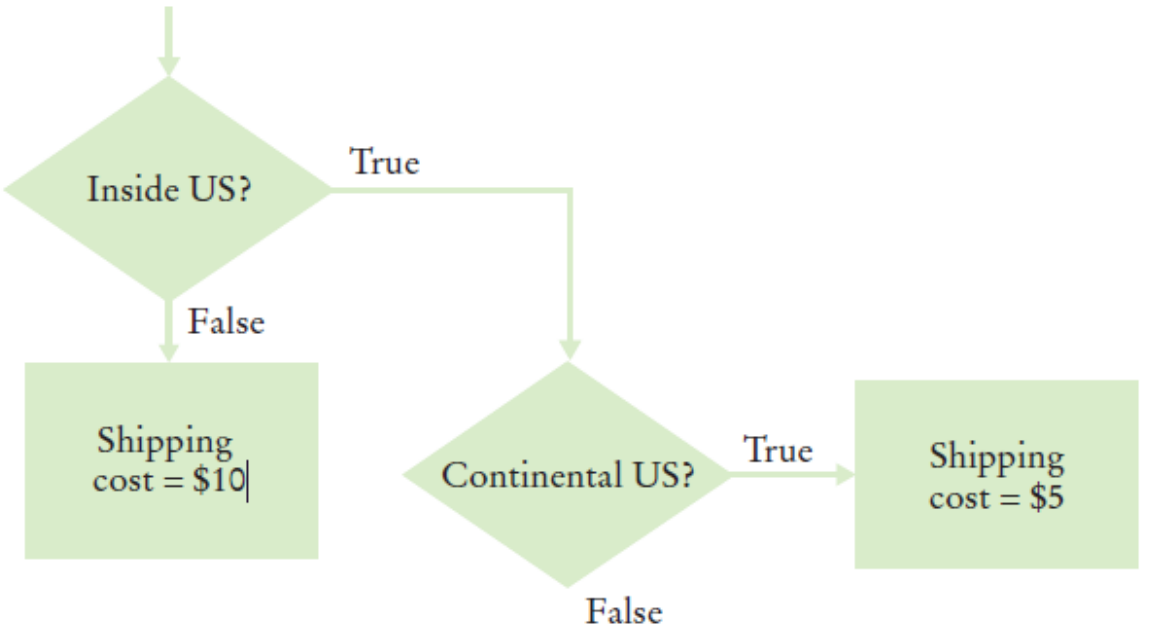
- Multiple Outcomes



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:



International Branch

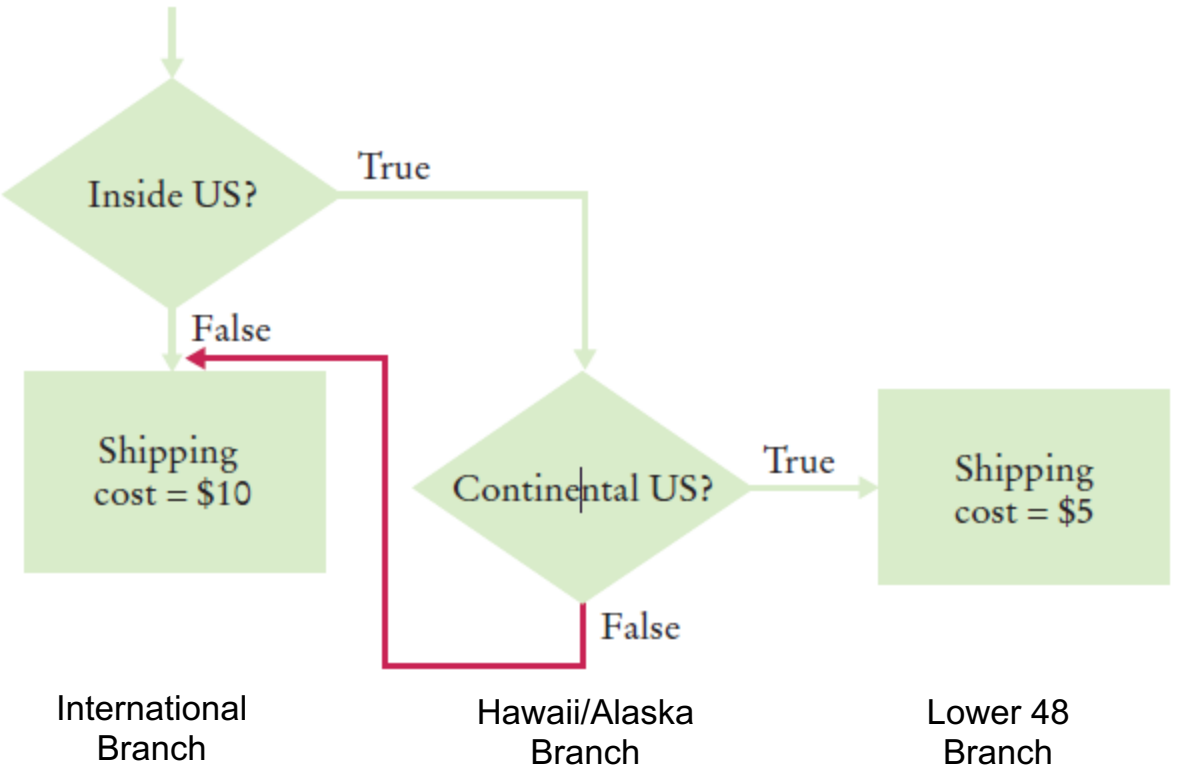
Hawaii/Alaska Branch

Lower 48 Branch

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.

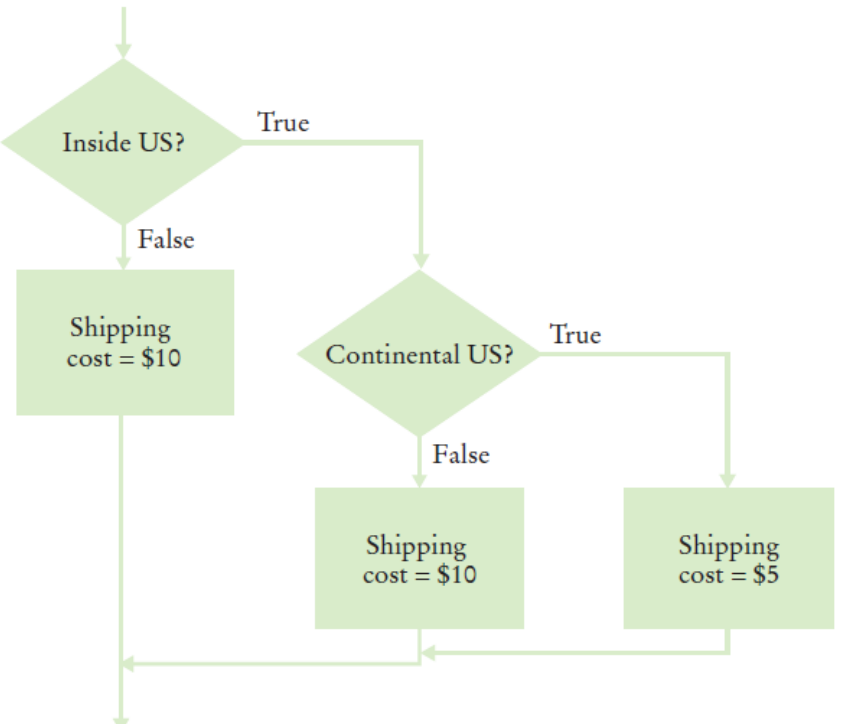
- **Do not do this!**



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.

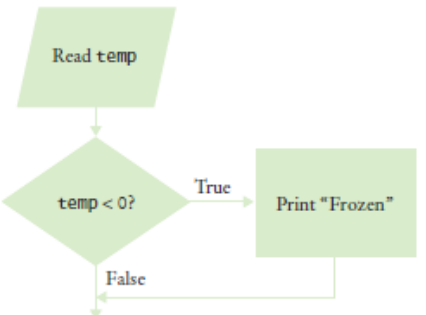
- Completed



Self Check 3.22

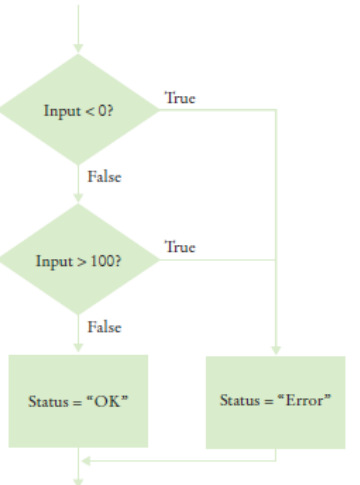
Draw a flowchart for a program that reads a value `temp` and prints “Frozen” if it is less than zero.

Answer:



Self Check 3.23

What is wrong with the flowchart at right?

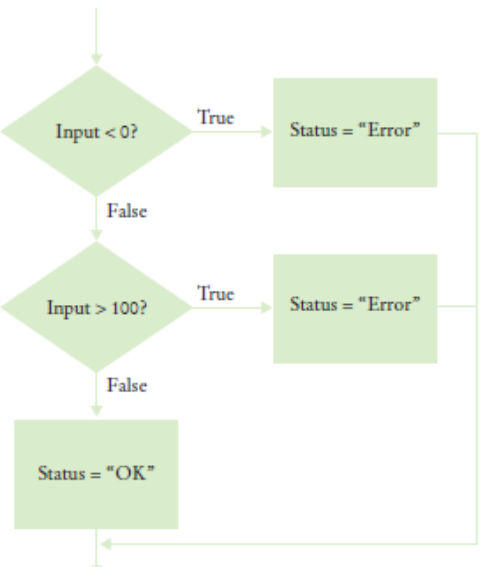


Answer: The "True" arrow from the first decision points into the "True" branch of the second decision, creating spaghetti code.

Self Check 3.24

How do you fix the flowchart of Self Check 23?

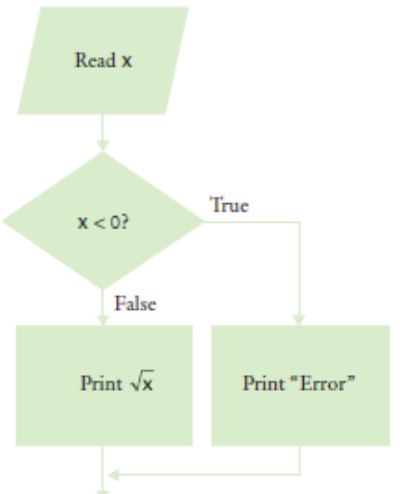
Answer: Here is one solution. In Section 3.7, you will see how you can combine the conditions for a more elegant solution.



Self Check 3.25

Draw a flowchart for a program that reads a value `x`. If it is less than zero, print “Error”. Otherwise, print its square root.

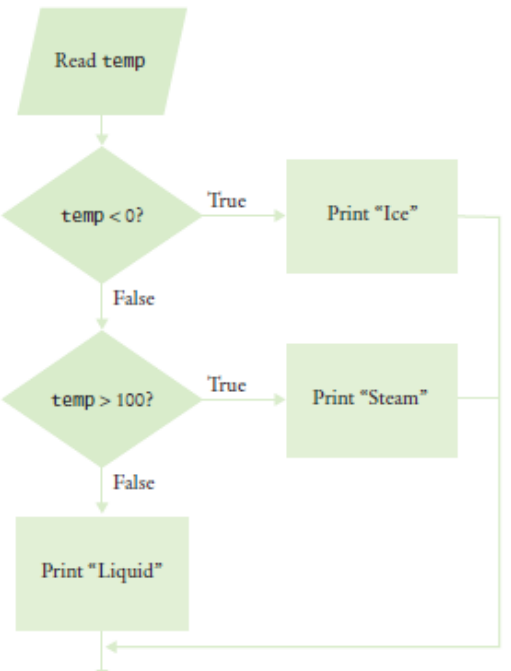
Answer:



Self Check 3.26

Draw a flowchart for a program that reads a value `temp`. If it is less than zero, print "Ice". If it is greater than 100, print "Steam". Otherwise, print "Liquid".

Answer:



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

Choosing Test Cases

- Choose input values that:
  - Test boundary cases and 0 values
  - A *boundary case* is a value that is tested in the code
  - Test each branch

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

Self Check 3.27

Using Figure 1 on page 85 as a guide, follow the process described in Section 3.6 to design a set of test cases for the `ElevatorSimulation.java` program in Section 3.1.

Answer:

Test Case	Expected Output	Comment
12	12	Below 13th floor
14	13	Above 13th floor
13	?	The specification is not clear— See Section 3.8 for a version of this program with error handling

Self Check 3.28

What is a boundary test case for the algorithm in How To 3.1 on page 95?
What is the expected output?

Answer: A boundary test case is a price of $128. A 16 percent discount should apply because the problem statement states that the larger discount applies if the price is *at least* $128. Thus, the expected output is $107.52.

Self Check 3.29

Using Figure 3 on page 99 as a guide, follow the process described in Section 3.6 to design a set of test cases for the `EarthquakeStrength.java` program in Section 3.3.

Answer:

Test Case	Expected Output	Comment
9	Most structures fall	
7.5	Many buildings destroyed	
6.5	Many buildings ...	
5	Damage to poorly...	
3	No destruction...	
8.0	Most structures fall	Boundary case. In this program, boundary cases are not as significant because the behavior of an earthquake changes gradually.
-1		The specification is not clear—see Self Check 16 for a version of this program with error handling.

Self Check 3.30

Suppose you are designing a part of a program for a medical robot that has a sensor returning an *x*- and *y*-location (measured in cm). You need to check whether the sensor location is inside the circle, outside the circle, or on the boundary (specifically, having a distance of less than 1 mm from the boundary). Assume the circle has center (0, 0) and a radius of 2 cm. Give a set of test cases.

Answer:

Test Case	Expected Output	Comment
(0.5, 0.5)	inside	
(4, 2)	outside	
(0, 2)	on the boundary	Exactly on the boundary
(1.414, 1.414)	on the boundary	Close to the boundary
(0, 1.9)	inside	Not less than 1 mm from the boundary
(0, 2.1)	outside	Not less than 1 mm from the boundary

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");
}
```

- 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 comparison, precede it with !

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

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

A	!A
true	false
false	true

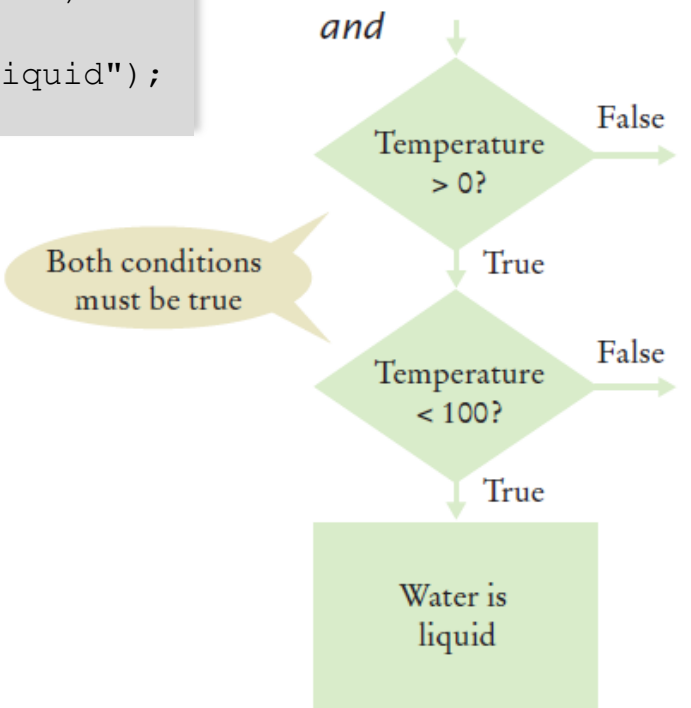
- If using !, try to use simpler logic:

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

and Flowchart

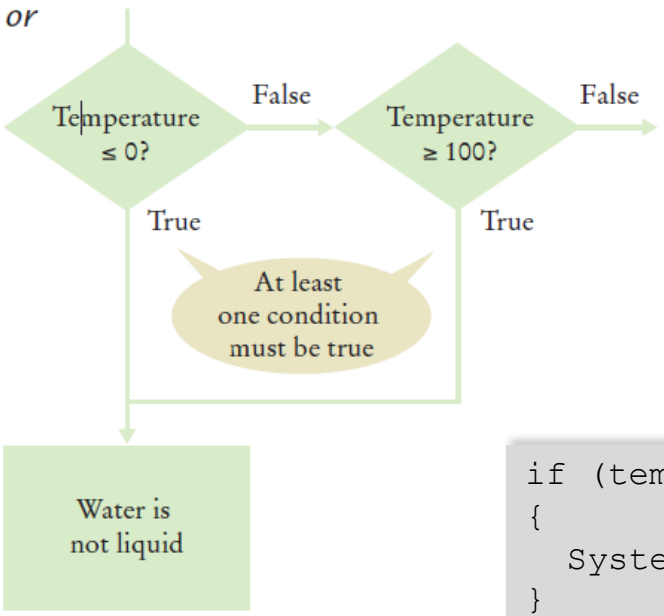
- This is often called ‘range checking’
  - Used to validate that input is between two values

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



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
`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).
`0 < x < 100`	**Error**	**Error:** This expression does not test whether `x` is between 0 and 100. The expression `0 < x` is a Boolean value. You cannot compare a Boolean value with the integer 100.
`x && y > 0`	**Error**	**Error:** This expression does not test whether `x` and `y` are positive. The left-hand side of `&&` is an integer, `x`, and the right-hand side, `y > 0`, is a Boolean value. You cannot use `&&` with an integer argument.
`!(0 < 200)`	`false`	`0 < 200` is `true`, therefore its negation is `false`.
`frozen == true`	`frozen`	There is no need to compare a Boolean variable with `true`.
`frozen == false`	`!frozen`	It is clearer to use `!` than to compare with `false`.

Self Check 3.31

Suppose x and y are two integers. How do you test whether both of them are zero?

Answer: `x == 0 && y == 0`

Self Check 3.32

How do you test whether at least one of them is zero?

Answer: `x == 0 || y == 0`

Self Check 3.33

How do you test whether *exactly one of them* is zero?

Answer: `(x == 0 && y != 0) || (y == 0 && x != 0)`

Self Check 3.34

What is the value of `!!frozen`?

Answer: The same as the value of `frozen`.

Self Check 3.35

What is the advantage of using the type `boolean` rather than strings "`false`"/"`true`" or integers 0/1?

Answer: You are guaranteed that there are no other values. With strings or integers, you would need to check that no values such as "`maybe`" or –1 enter your calculations.

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)
```

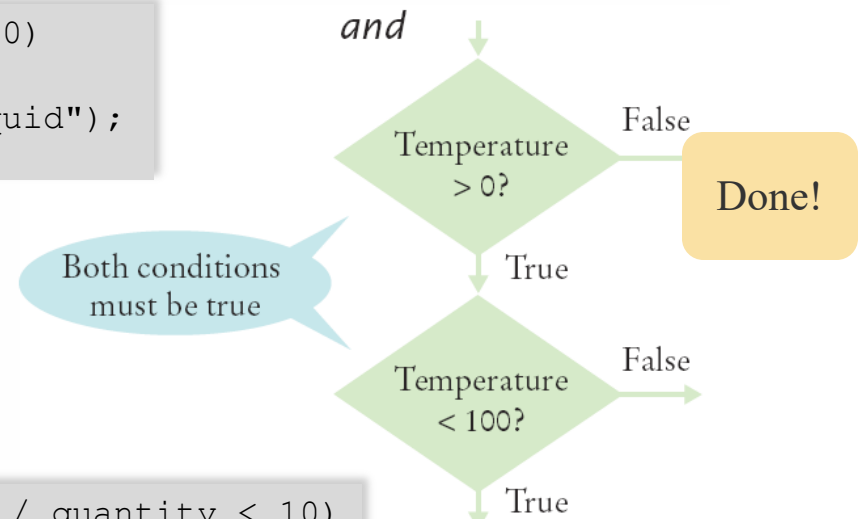
Common Error

- 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");
}
```



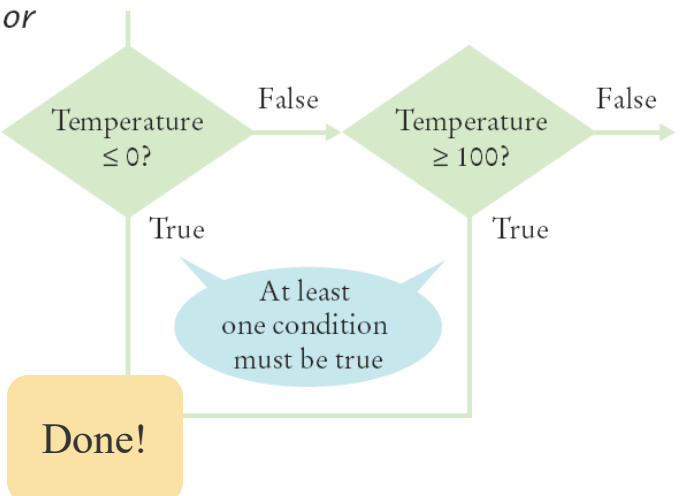

```
if (quantity > 0 && price / quantity < 10)
```

Short-Circuit Evaluation: ||

- If the left half of the *or* is true, why look further?
- Java doesn't!
- Don't do these second:
  - Assignment
  - Output

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

or



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.

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)
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.
```

Self Check 3.36

In the `ElevatorSimulation2` program, what is the output when the input is

a. 100?
b. –1?
c. 20?
d. thirteen?

Answer:

(a)
```
Error: The floor must be between 1 and 20.
```
(b)
```
Error: The floor must be between 1 and 20.
```
(c)
```
19
```
(d)
```
Error: Not an integer.
```

Self Check 3.37

Your task is to rewrite lines 19–26 of the `ElevatorSimulation2` program so that there is a single `if` statement with a complex condition. What is the condition?

```
if (. . .)
{
   System.out.println("Error: Invalid floor number");
}
```

Answer: `floor == 13 || floor <= 0 || floor > 20`

Self Check 3.38

In the Sherlock Holmes story “The Adventure of the Sussex Vampire”, the inimitable detective uttered these words: “Matilda Briggs was not the name of a young woman, Watson, … It was a ship which is associated with the giant rat of Sumatra, a story for which the world is not yet prepared.” Over a hundred years later, researchers found giant rats in Western New Guinea, another part of Indonesia.

Suppose you are charged with writing a program that processes rat weights. It contains the statements

```
System.out.print("Enter weight in kg: ");
double weight = in.nextDouble();
```

What input checks should you supply?

Self Check 3.38

Answer: Check for `in.hasNextDouble()`, to make sure a researcher didn't supply an input such as `oh my`. Check for `weight <= 0`, because any rat must surely have a positive weight. We don't know how giant a rat could be, but the New Guinea rats weighed no more than 2 kg. A regular house rat (*rattus rattus*) weighs up to 0.2 kg, so we'll say that any weight > 10 kg was surely an input error, perhaps confusing grams and kilograms. Thus, the checks are

```
if (in.hasNextDouble())
{
   double weight = in.nextDouble();
   if (weight <= 0)
   {
      System.out.println(
         "Error: Weight must be positive.");
   }
   else if (weight > 10)
   {
      System.out.println( "Error: Weight > 10 kg.");
   }
   else
   {
       Process valid weight.
   }
}
else
}
   System.out.print("Error: Not a number");
}
```

Self Check 3.39

Run the following test program and supply inputs 2 and three at the prompts. What happens? Why?

```
import java.util.Scanner
public class Test
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      System.out.print("Enter an integer: ");
      int m = in.nextInt();
      System.out.print("Enter another integer: ");
      int n = in.nextInt();
      System.out.println(m + " " + n);
   }
}
```

Answer: The second input fails, and the program terminates without printing anything.