Chapter 3 - Decisions

APROG – Algoritmia e Programação

Emanuel Cunha Silva

ecs@isep.ipp.pt

Bibliography:
Big Java Late Objects
Book by Cay S. Horstmann

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

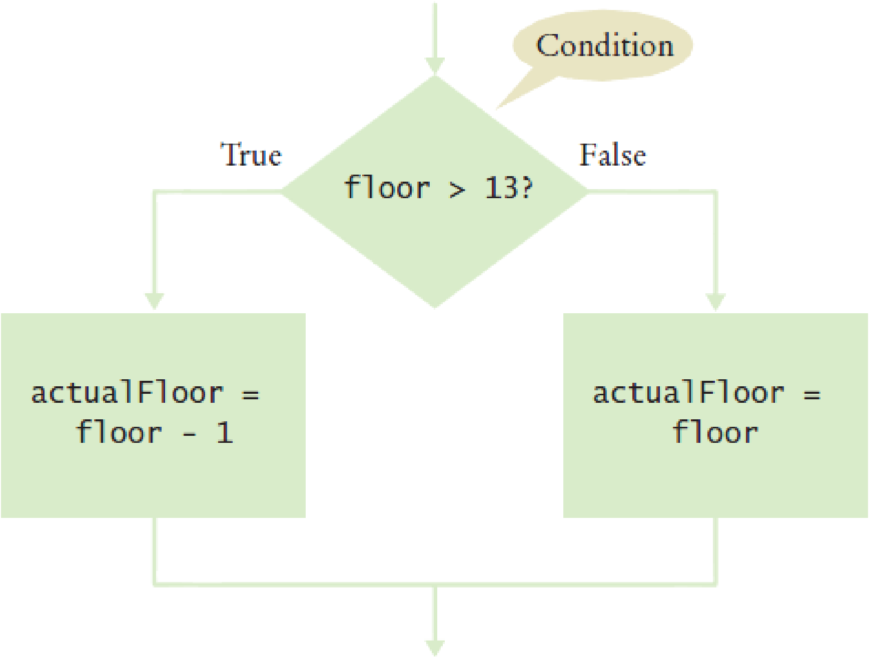
The `if` Statement

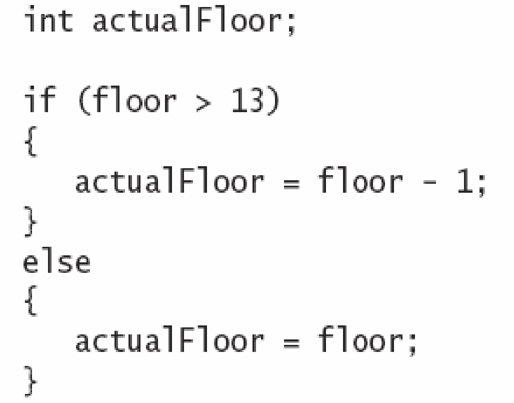
- A computer program often needs to make decisions based on input, or circumstances
- For example, buildings often ‘skip’ the 13th floor, and elevators should too
  - The 14th floor is really the 13th floor
  - So every floor above 12 is really ‘floor – 1’
    - If floor > 12, Actual floor = floor -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

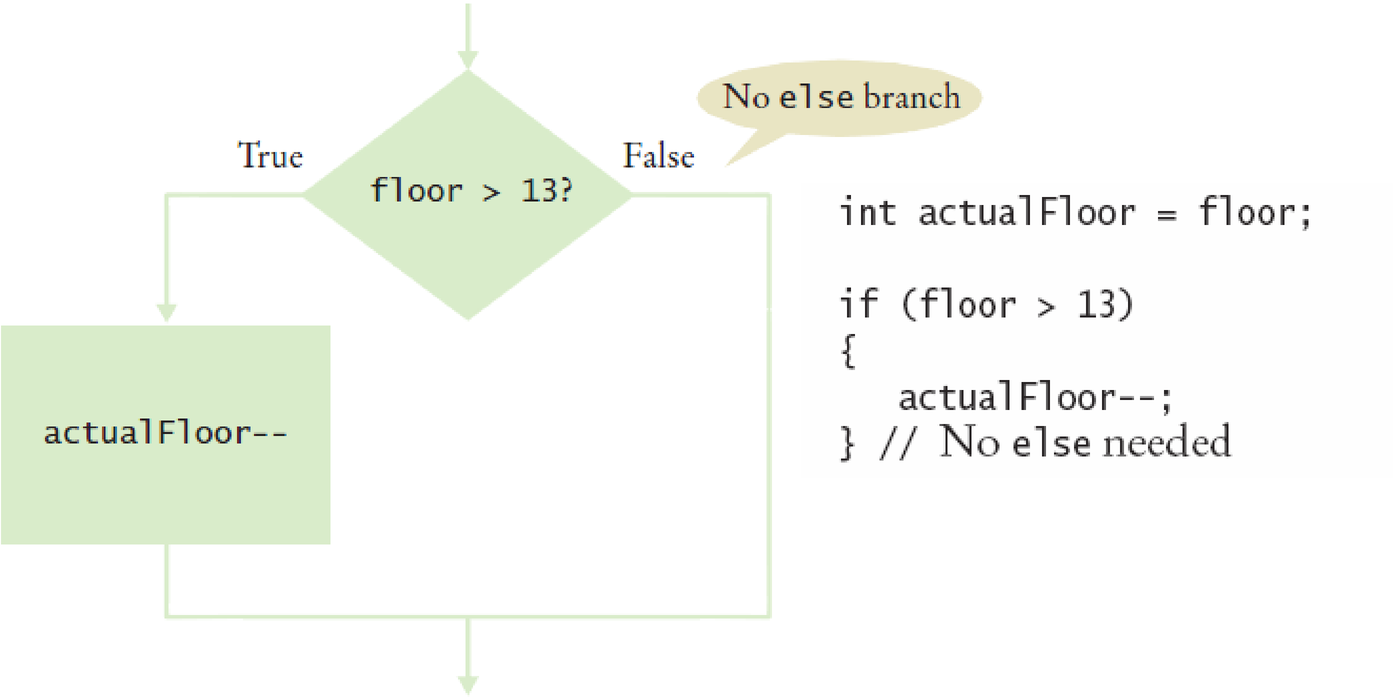




```
int actualFloor;

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

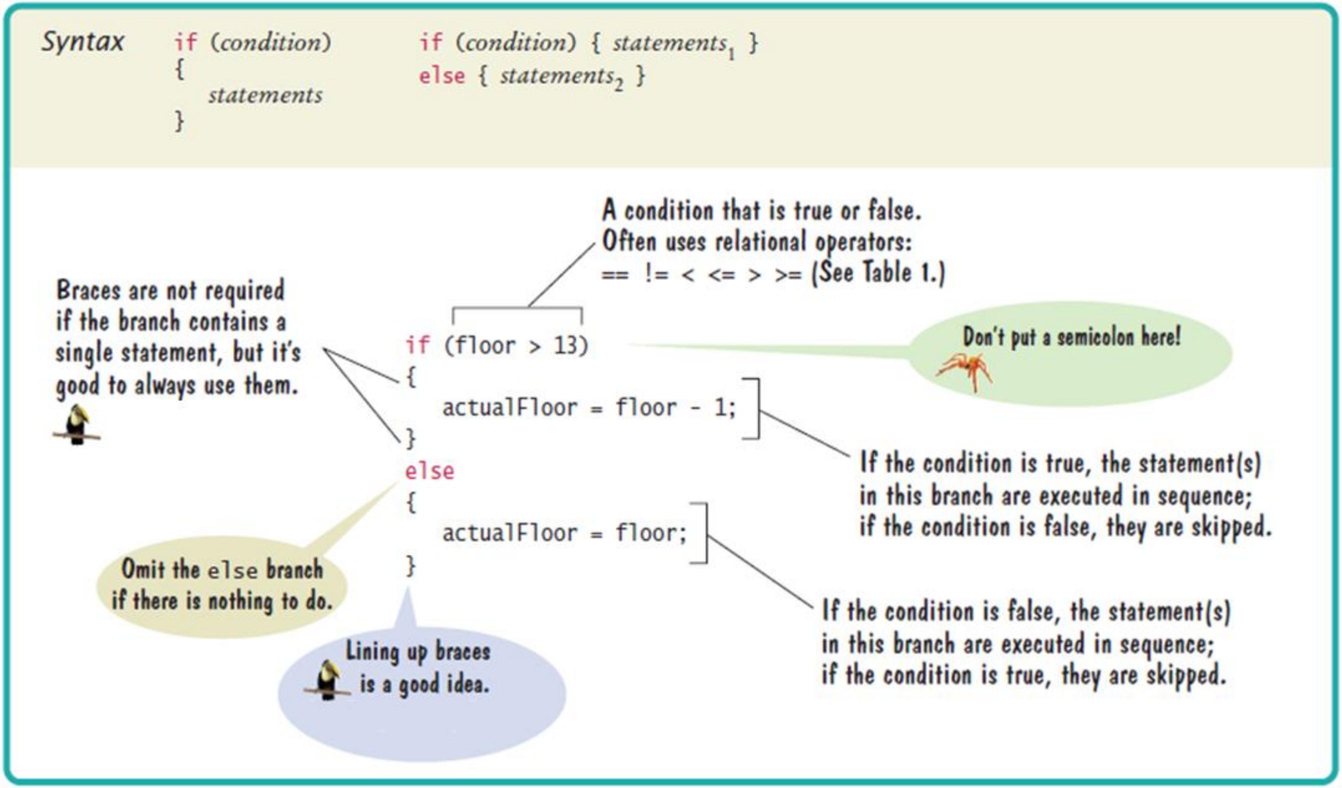
Flowchart with only a True Branch

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



```
int actualFloor = floor;

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

The if Statement



Syntax

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

if (*condition*) { *statements*$_1$ }
else { *statements*$_2$ }

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

A condition that is true or false. Often uses relational operators: == != < <= > >= (See Table 1.)

Don't put a semicolon here!

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

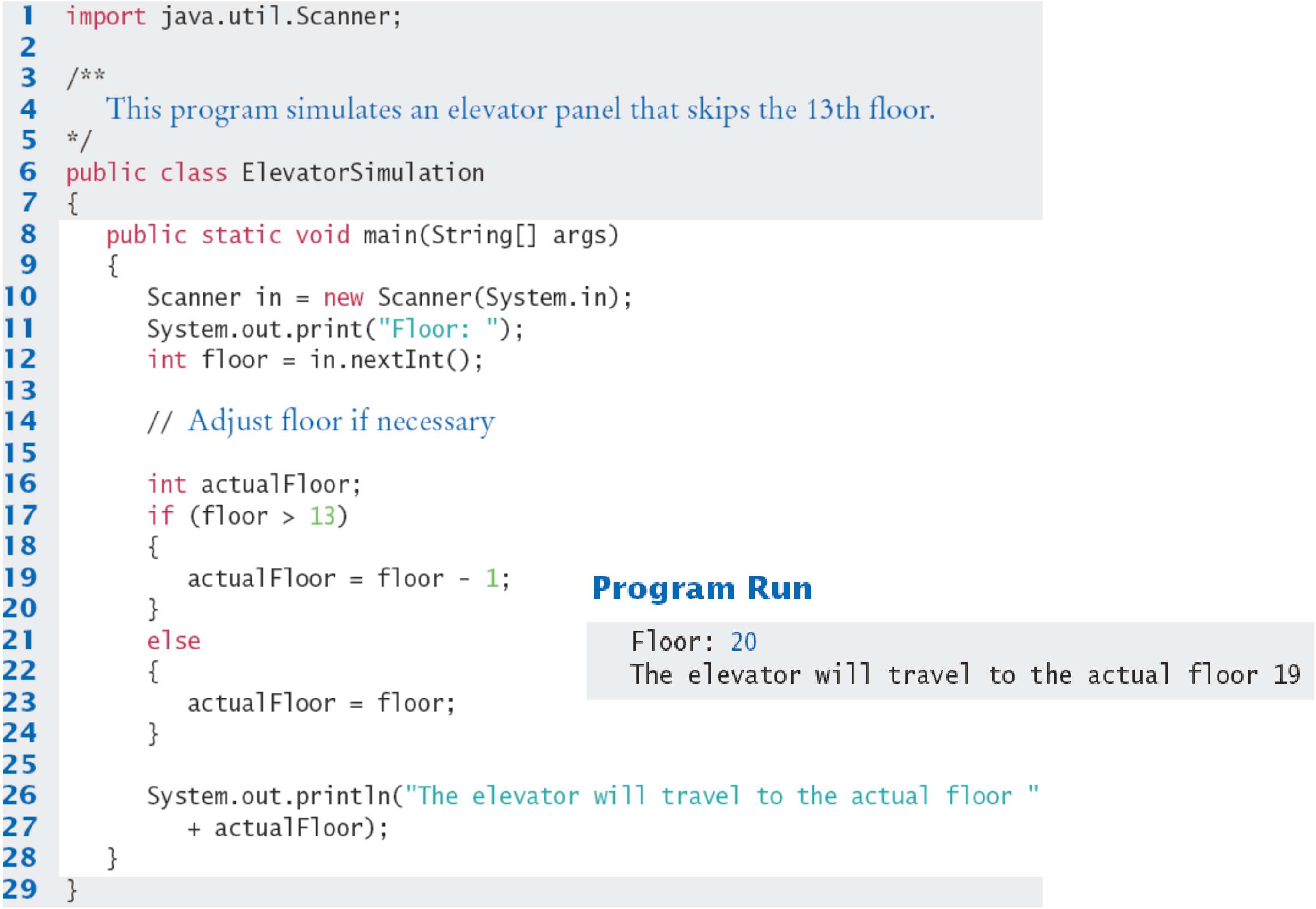
If the condition is true, the statement(s) in this branch are executed in sequence; if the condition is false, they are skipped.

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.

ElevatorSimulation.java



```java
import java.util.Scanner;

/**
   This program simulates an elevator panel that skips the 13th floor.
*/
public class ElevatorSimulation
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      System.out.print("Floor: ");
      int floor = in.nextInt();

      // Adjust floor if necessary

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

      System.out.println("The elevator will travel to the actual floor "
         + actualFloor);
   }
}
```

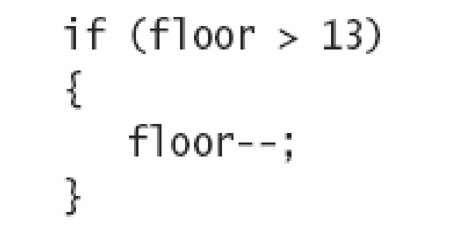
Program Run

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

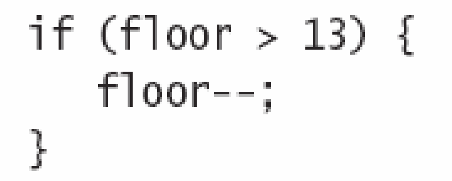
Tips On Using Braces

- Line up all pairs of braces vertically
  - Lined up

  Not aligned (saves lines)

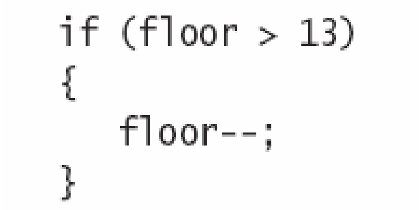


```
if (floor > 13)
{
   floor--;
}
```

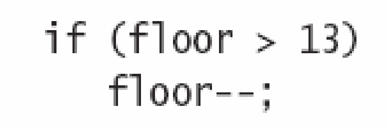


```
if (floor > 13) {
   floor--;
}
```

- Always use braces
  - Although single statement clauses do not require them



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

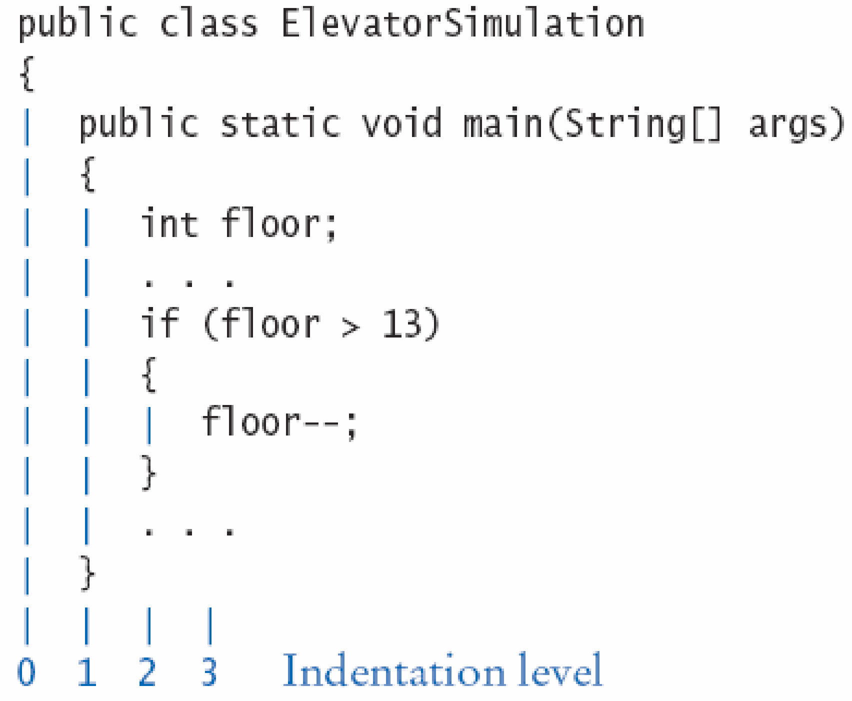


```
if (floor > 13)
   floor--;
```

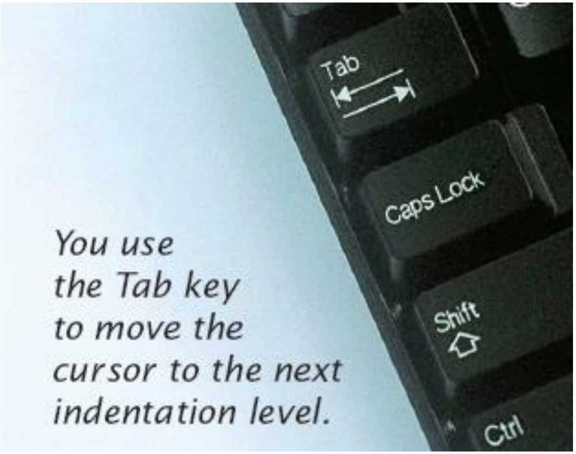
- Most programmer’s editors have a tool to align matching braces.

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
```



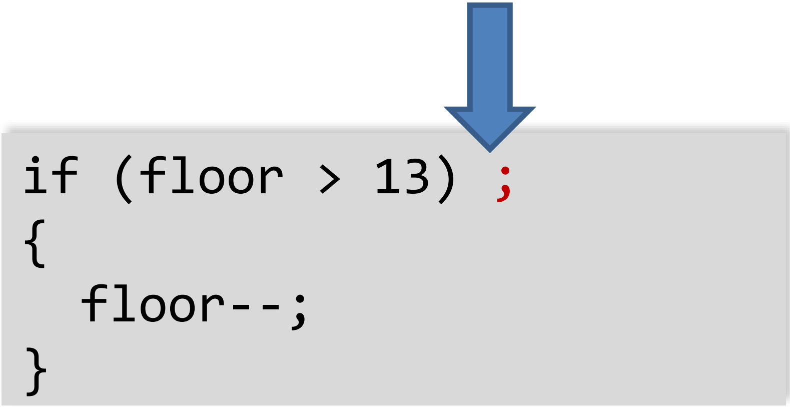
You use
the Tab key
to move the
cursor to the next
indentation level.

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

Common Error

- A semicolon after an `if` statement

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



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

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

The Conditional Operator

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

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

  - 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!

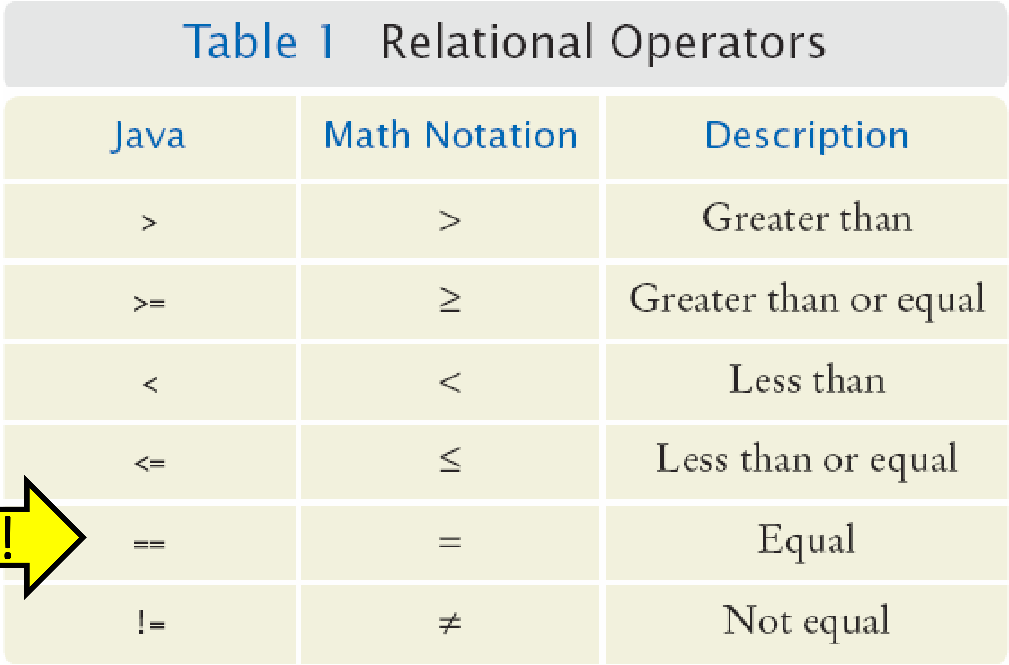
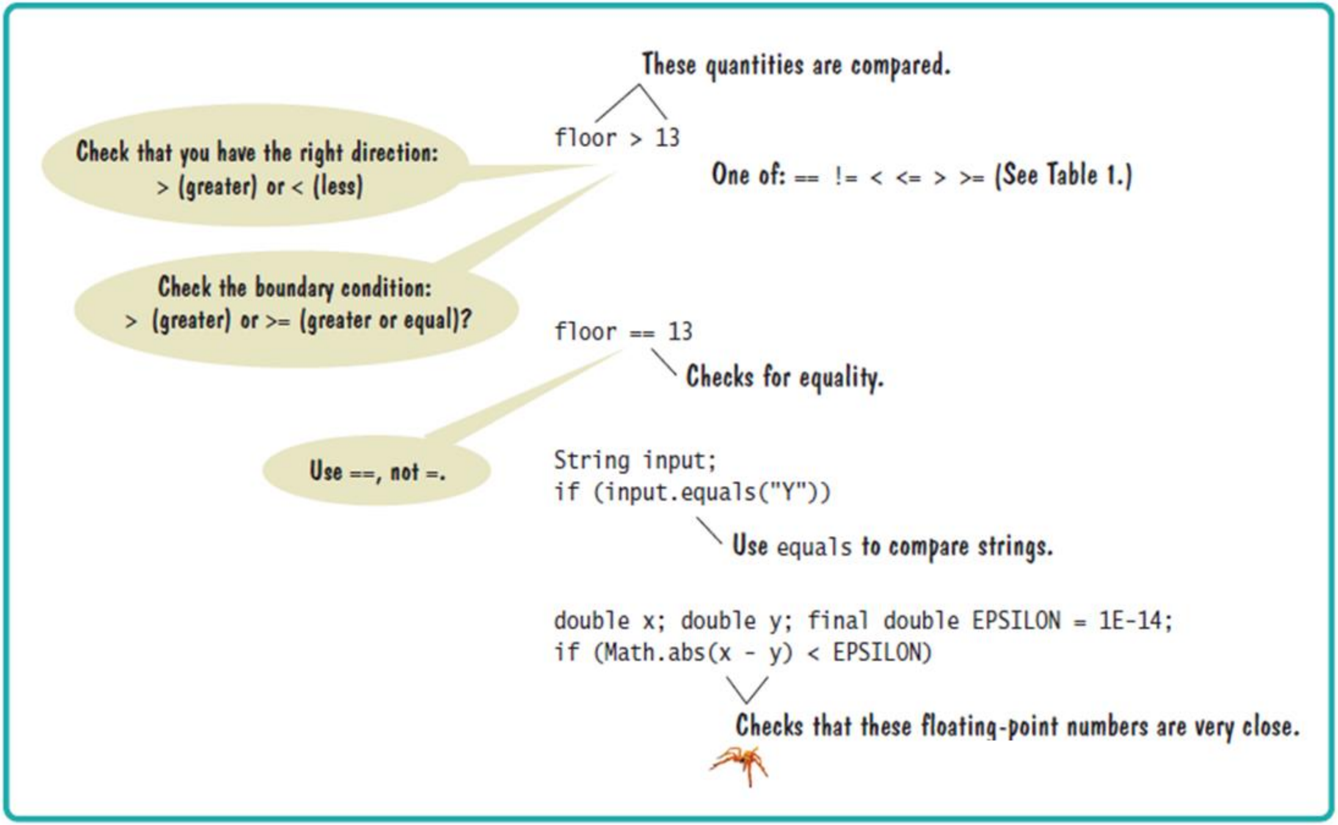


Table 1 Relational Operators

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

Comparisons



These quantities are compared.

```
floor > 13
```

One of: == != < <= > >= (See Table 1.)

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 =.

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

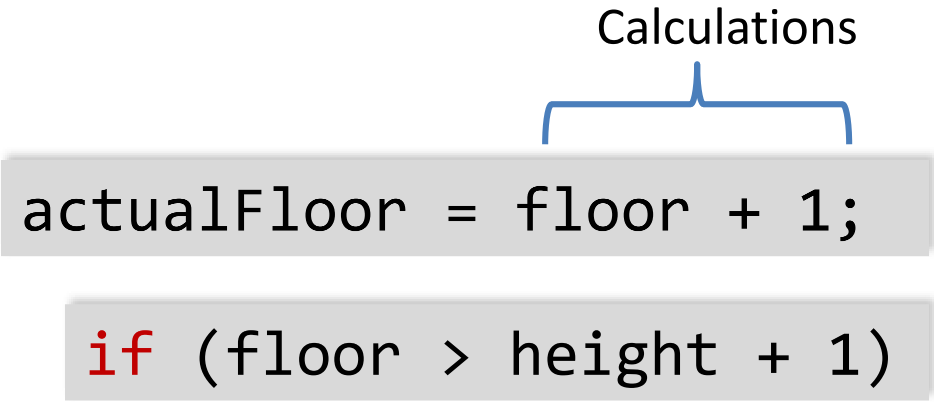
Use equals to compare strings.

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

Checks that these floating-point numbers are very close.

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

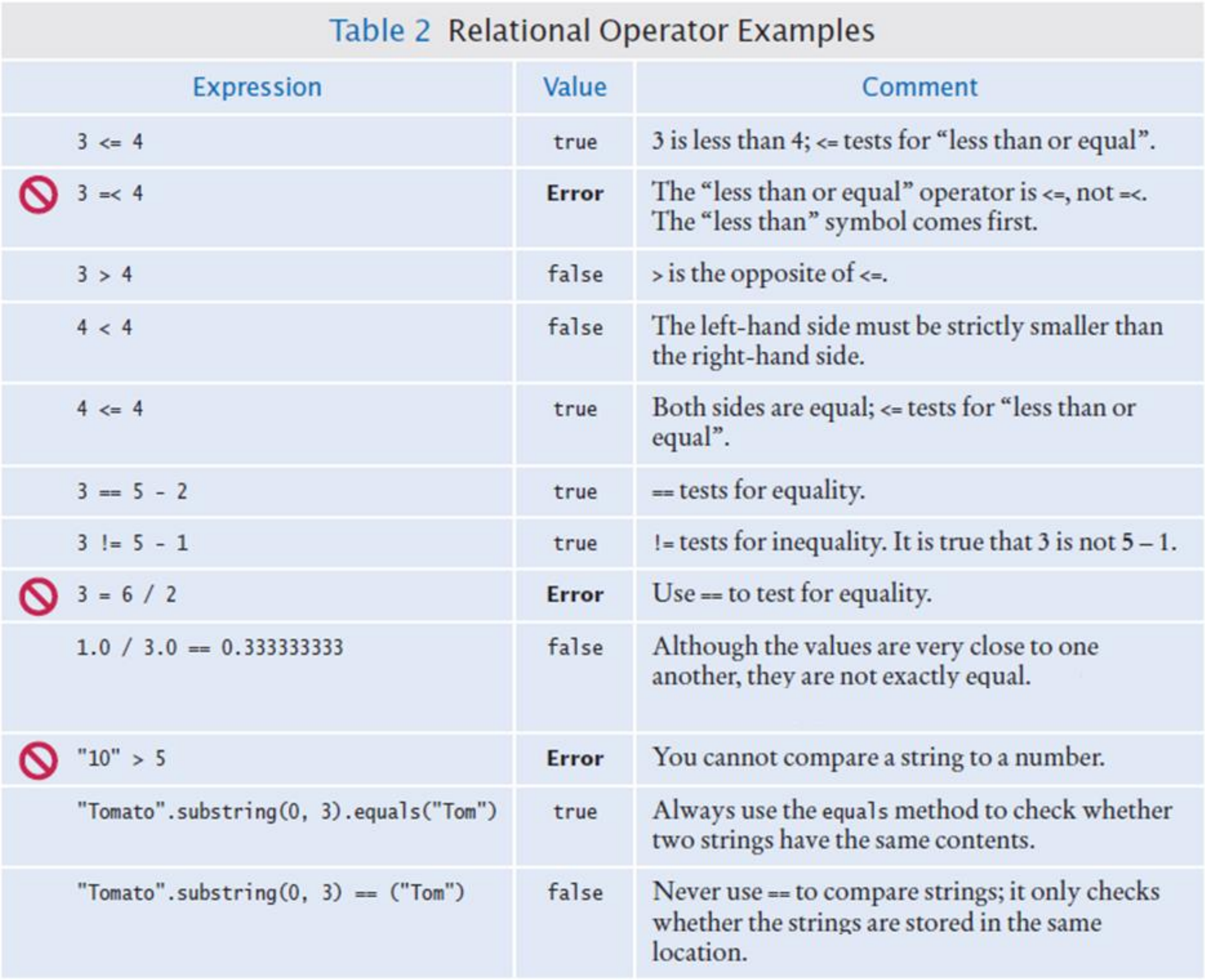


Table 2 Relational Operator Examples

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.
⊘ `"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.

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

What is the error in this statement?

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

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

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")`

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 2.00000000000000044
```

The Use of EPSILON

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

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

```
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

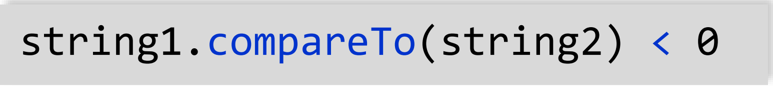
- Java creates a new String every time a new word inside double-quotes is used
    - If there is one that matches it exactly, Java re-uses it

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

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

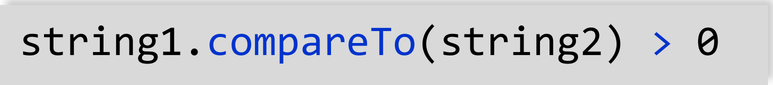
Lexicographical Order

- To compare Strings in ‘dictionary’ order
  - When compared using `compareTo`, `string1` comes:
    - Before `string2` if



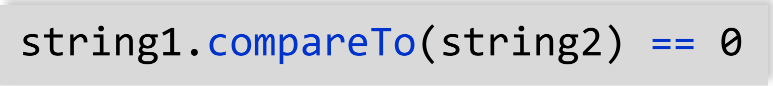
```
string1.compareTo(string2) < 0
```

    - After `string2` if



```
string1.compareTo(string2) > 0
```

    - Equal to `string2` if

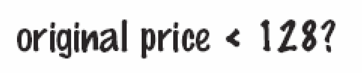


```
string1.compareTo(string2) == 0
```

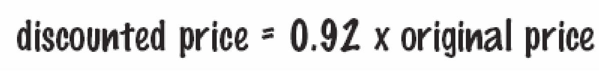
  - Notes
    - All UPPERCASE letters come before lowercase
    - ‘space’ comes before all other printable characters
    - Digits (0-9) come before all letters
    - See Appendix A for the Basic Latin Unicode (ASCII) table

Implementing an `if` Statement

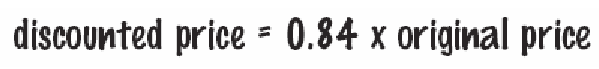
1) Decide on a branching condition


original price < 128?

2) Write pseudocode for the true branch


discounted price = 0.92 x original price

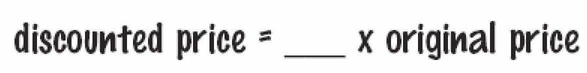
3) Write pseudocode for the false branch


discounted price = 0.84 x original price

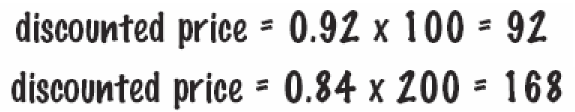
4) Double-check relational operators

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

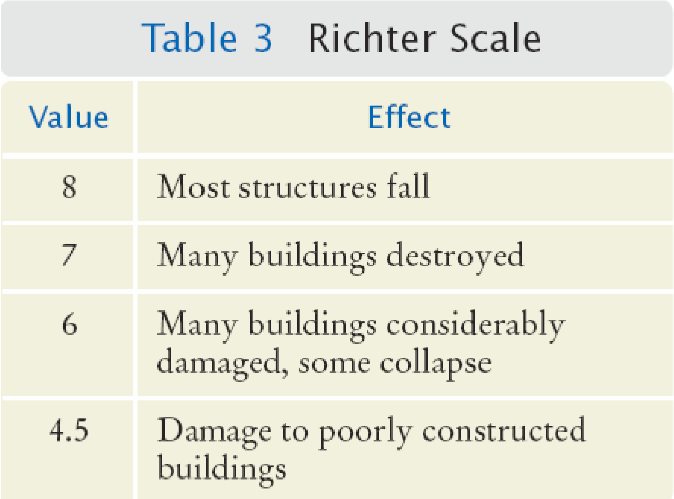
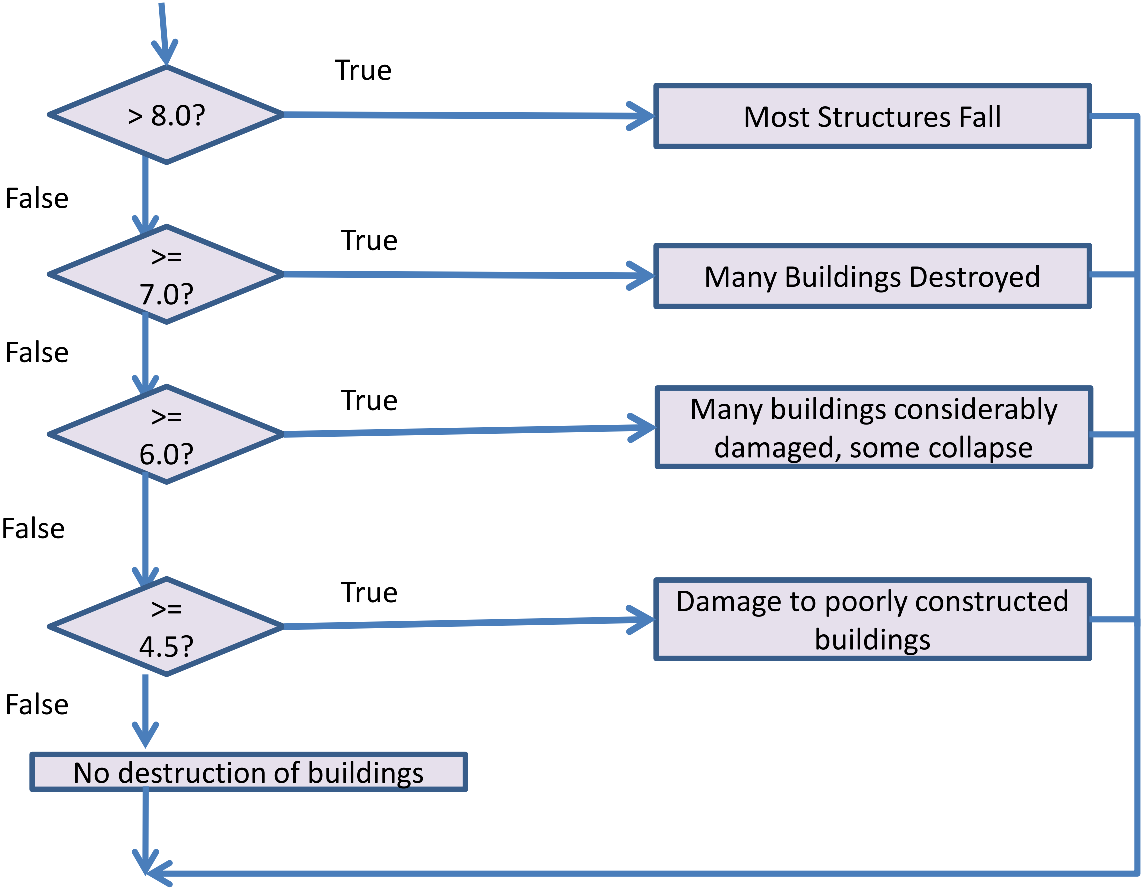


Table 3 Richter Scale

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


True
> 8.0?
Most Structures Fall
False
True
>=
7.0?
Many Buildings Destroyed
False
True
>=
6.0?
Many buildings considerably damaged, some collapse
False
True
>=
4.5?
Damage to poorly constructed buildings
False
No destruction of buildings

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

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

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

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.

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.

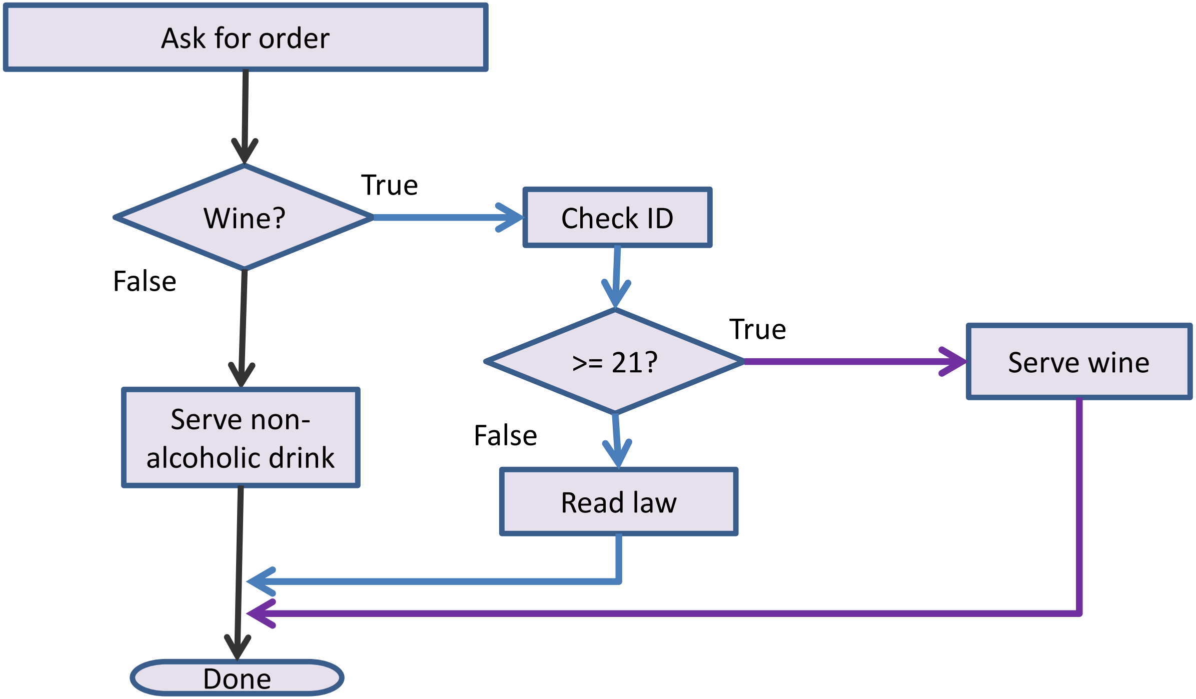
```
int digit = . . .;
switch (digit)
{
  case 1: digitName = "one";   break;
  case 2: digitName = "two";   break;
  case 3: digitName = "three"; break;
  case 4: digitName = "four";  break;
  case 5: digitName = "five";  break;
  case 6: digitName = "six";   break;
  case 7: digitName = "seven"; break;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

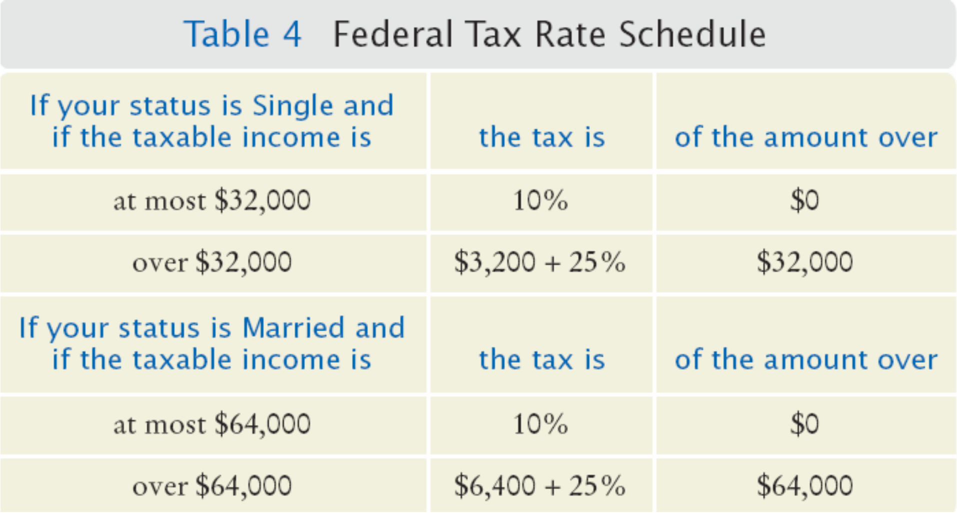
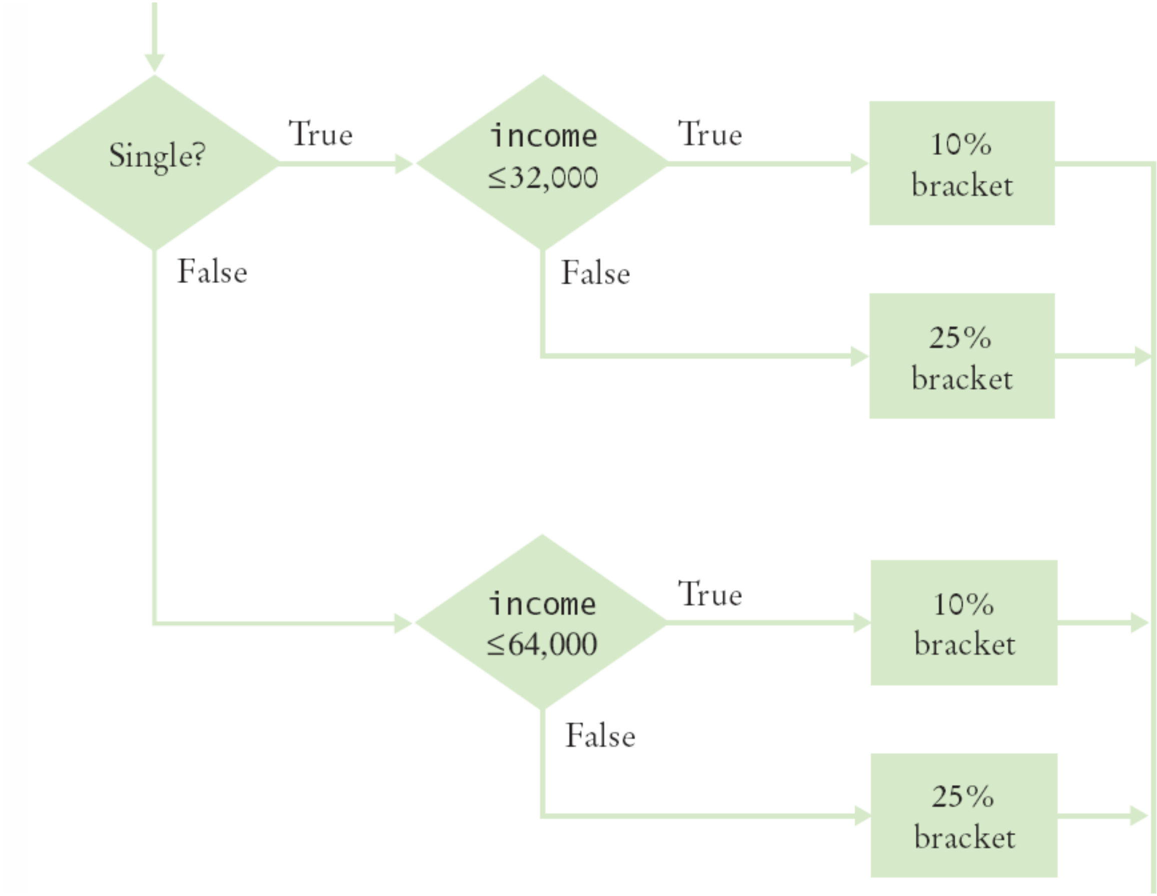


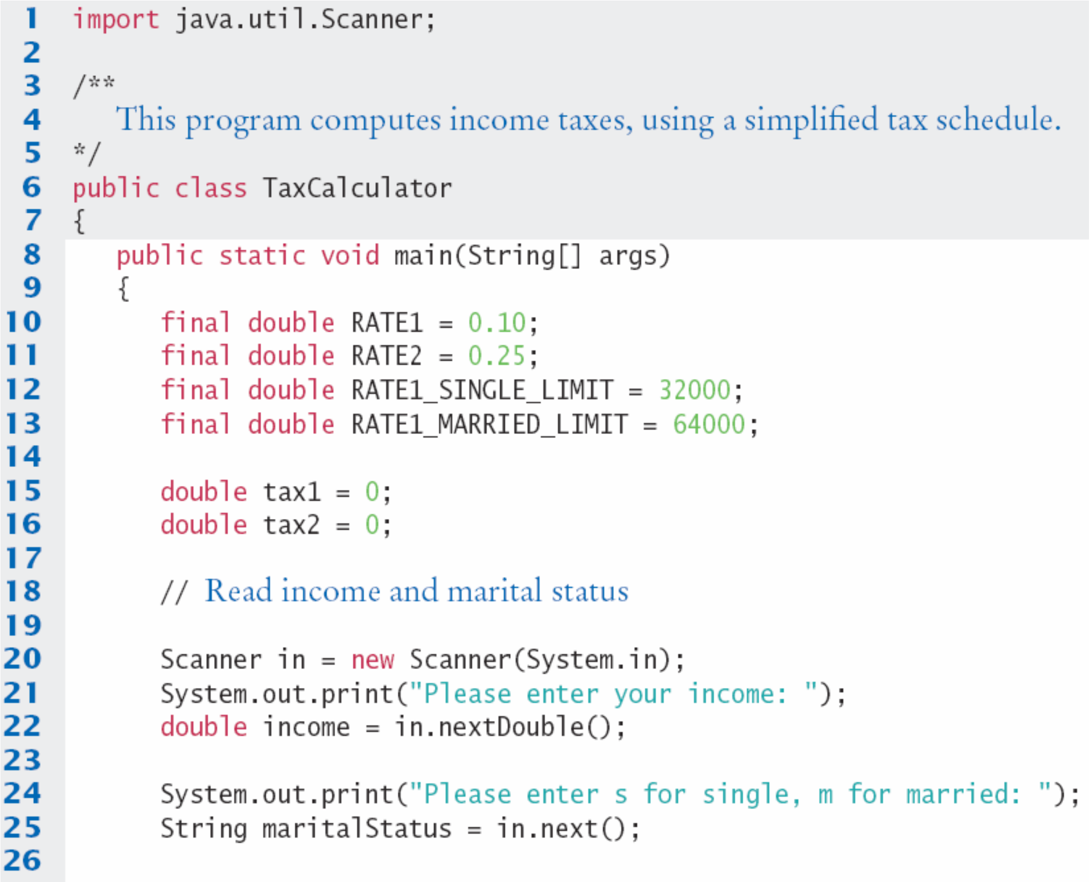
Table 4 Federal Tax Rate Schedule

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



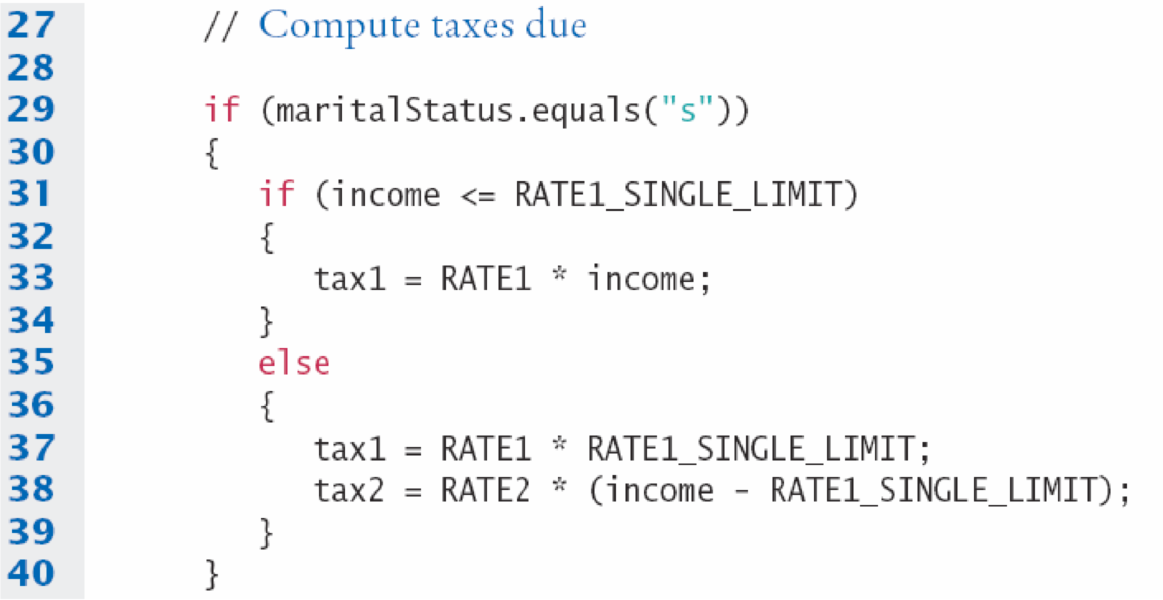
TaxCalculator.java (1)



```java
import java.util.Scanner;

/**
   This program computes income taxes, using a simplified tax schedule.
*/
public class TaxCalculator
{
   public static void main(String[] args)
   {
      final double RATE1 = 0.10;
      final double RATE2 = 0.25;
      final double RATE1_SINGLE_LIMIT = 32000;
      final double RATE1_MARRIED_LIMIT = 64000;

      double tax1 = 0;
      double tax2 = 0;

      // Read income and marital status

      Scanner in = new Scanner(System.in);
      System.out.print("Please enter your income: ");
      double income = in.nextDouble();

      System.out.print("Please enter s for single, m for married: ");
      String maritalStatus = in.next();

```

TaxCalculator.java (2)

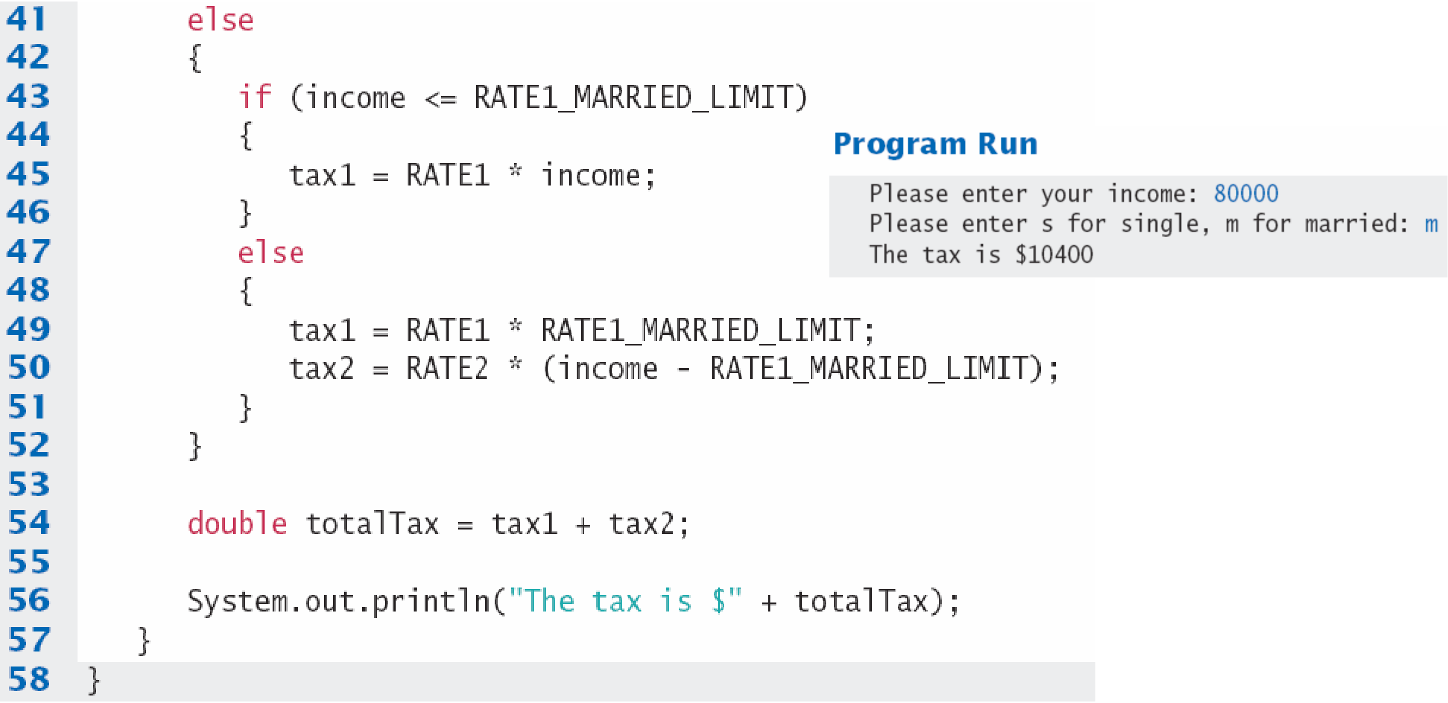
- The ‘True’ branch (Married)
  - Two branches within this branch



```
      // Compute taxes due

      if (maritalStatus.equals("s"))
      {
         if (income <= RATE1_SINGLE_LIMIT)
         {
            tax1 = RATE1 * income;
         }
         else
         {
            tax1 = RATE1 * RATE1_SINGLE_LIMIT;
            tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT);
         }
      }
```

TaxCalculator.java (3)

- The ‘False’ branch (Not married)



```java
      else
      {
         if (income <= RATE1_MARRIED_LIMIT)
         {
            tax1 = RATE1 * income;
         }
         else
         {
            tax1 = RATE1 * RATE1_MARRIED_LIMIT;
            tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
         }
      }

      double totalTax = tax1 + tax2;

      System.out.println("The tax is $" + totalTax);
   }
}
```

Program Run

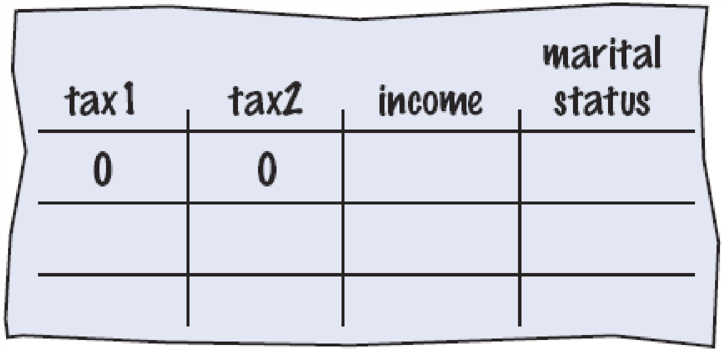
```
Please enter your income: 80000
Please enter s for single, m for married: m
The tax is $10400
```

Hand-Tracing

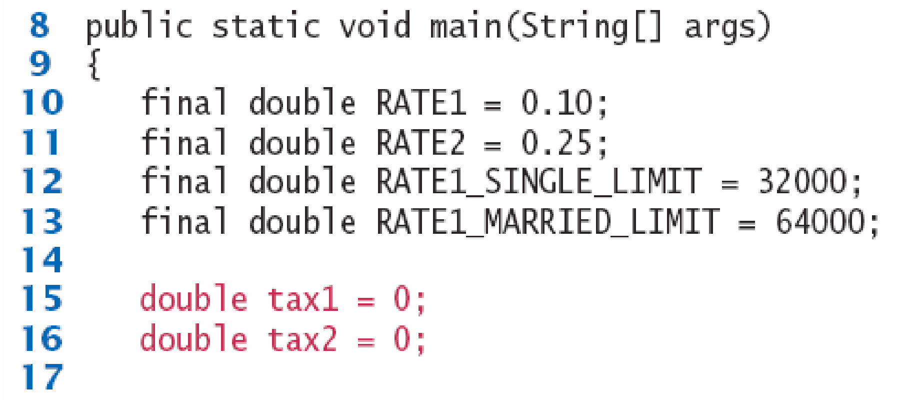
- Hand-tracing helps you understand whether a program works correctly
- Create a table of key variables
    - Use pencil and paper to track their values
- Works with pseudocode or code
    - Track location with a marker such as a paper clip
- Use example input values that:
    - You know what the correct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

Hand-Tracing Tax Example (1)

- Setup
    - Table of variables
    - Initial values



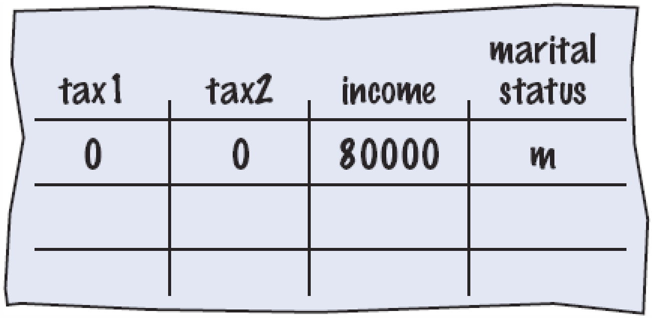
tax1	tax2	income	marital status
0	0		



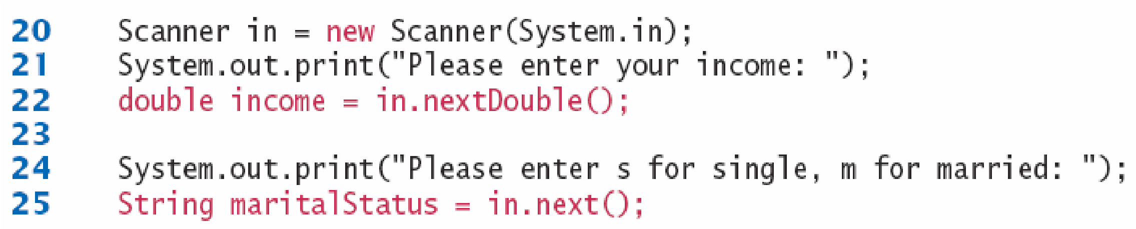
```
public static void main(String[] args)
{
    final double RATE1 = 0.10;
    final double RATE2 = 0.25;
    final double RATE1_SINGLE_LIMIT = 32000;
    final double RATE1_MARRIED_LIMIT = 64000;

    double tax1 = 0;
    double tax2 = 0;

```

Hand-Tracing Tax Example (2)

- Input variables
  - From user
  - Update table

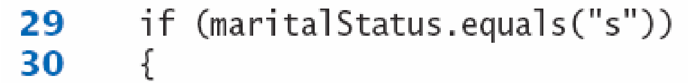


tax1	tax2	income	marital status
0	0	80000	m

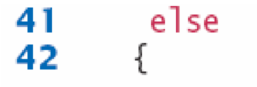


```
20     Scanner in = new Scanner(System.in);
21     System.out.print("Please enter your income: ");
22     double income = in.nextDouble();
23
24     System.out.print("Please enter s for single, m for married: ");
25     String maritalStatus = in.next();
```

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



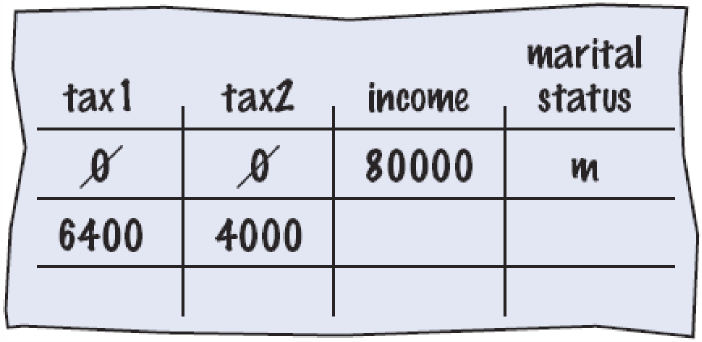
```
29     if (maritalStatus.equals("s"))
30     {
```



```
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

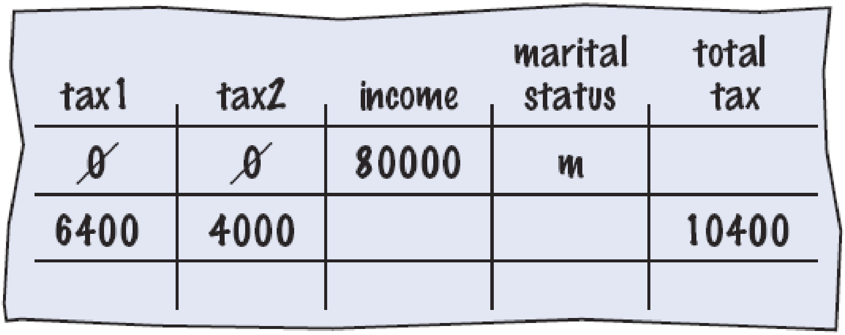


tax1	tax2	income	marital status
~~0~~	~~0~~	80000	m
6400	4000		

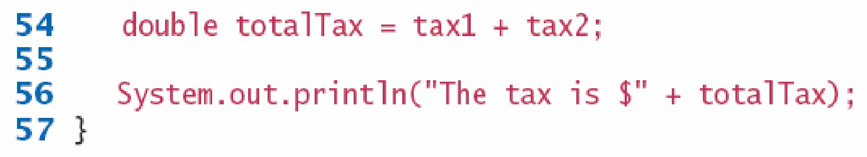
```
43        if (income <= RATE1_MARRIED_LIMIT)
44        {
45           tax1 = RATE1 * income;
46        }
47        else
48        {
49           tax1 = RATE1 * RATE1_MARRIED_LIMIT;
50           tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
51        }
```

Hand-Tracing Tax Example (4)

- Output
    - Calculate
    - As expected?



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



```
    double totalTax = tax1 + tax2;

    System.out.println("The tax is $" + totalTax);
}
```

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
```

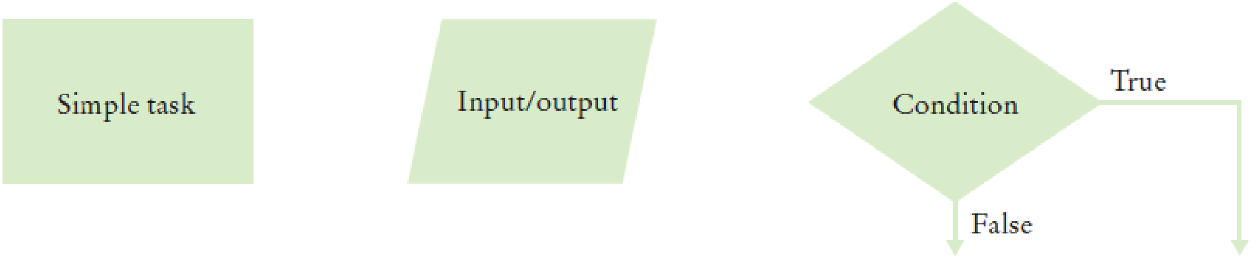
  - The indentation level suggests that the `else` is related to the `if` country (“USA”)
    - Else clauses always associate to the closest `if`

Problem Solving: Flowcharts

- You have see a few basic flowcharts

- A flowchart shows the structure of decisions and tasks to solve a problem

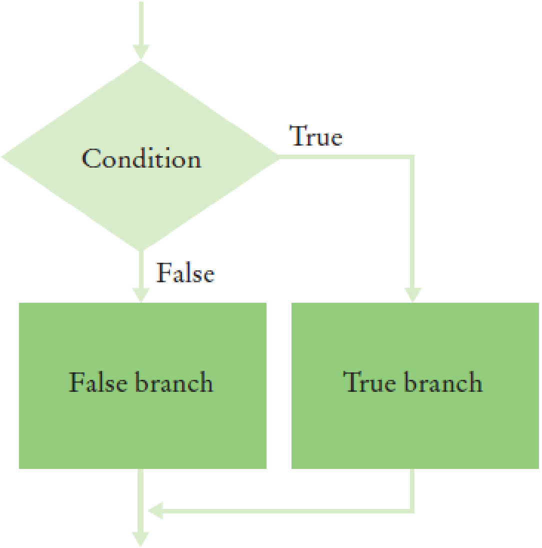
- Basic flowchart elements:



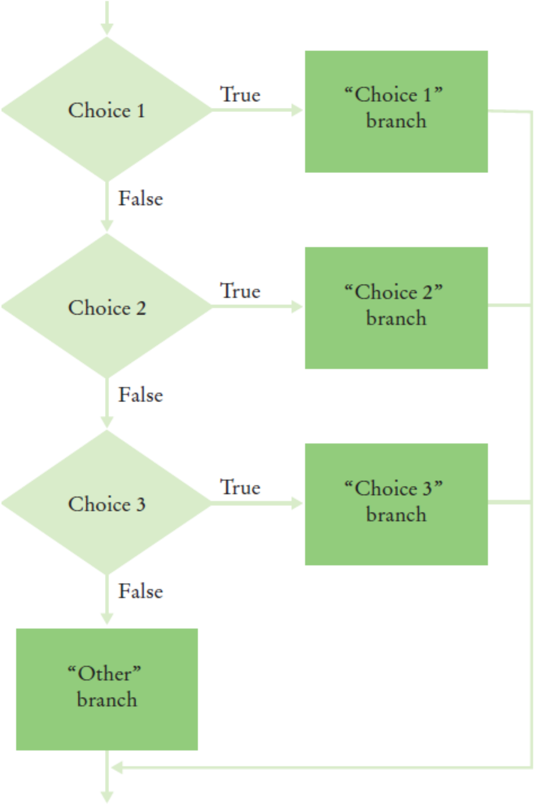
- Connect them with arrows
    - But never point an arrow
    - inside another branch!

Conditional Flowcharts

- Two Outcomes



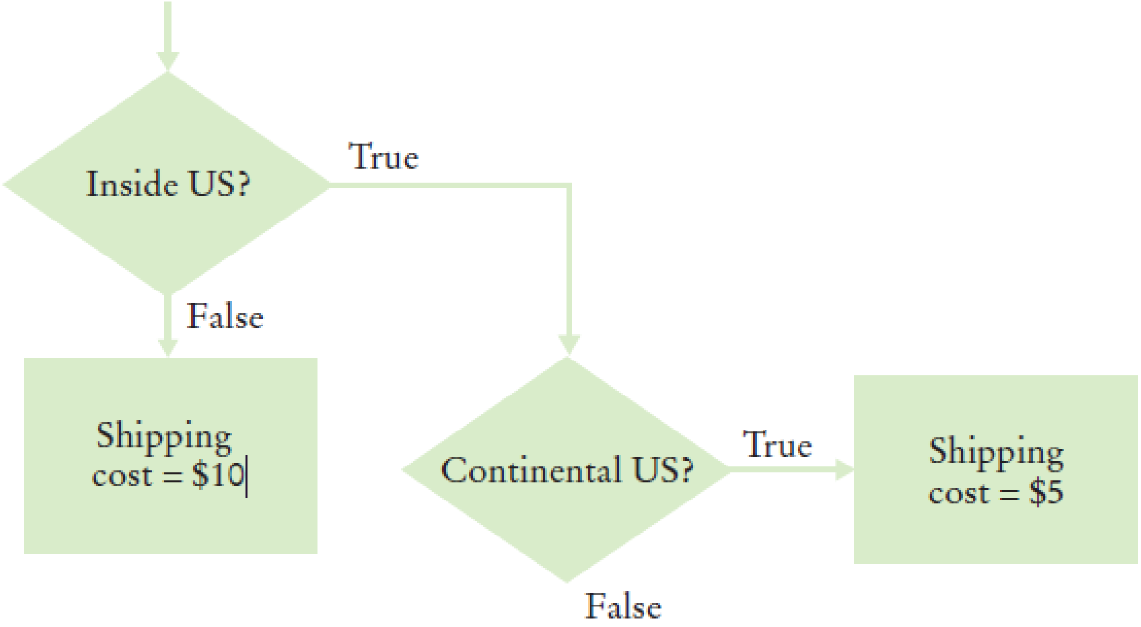
- Multiple Outcomes



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

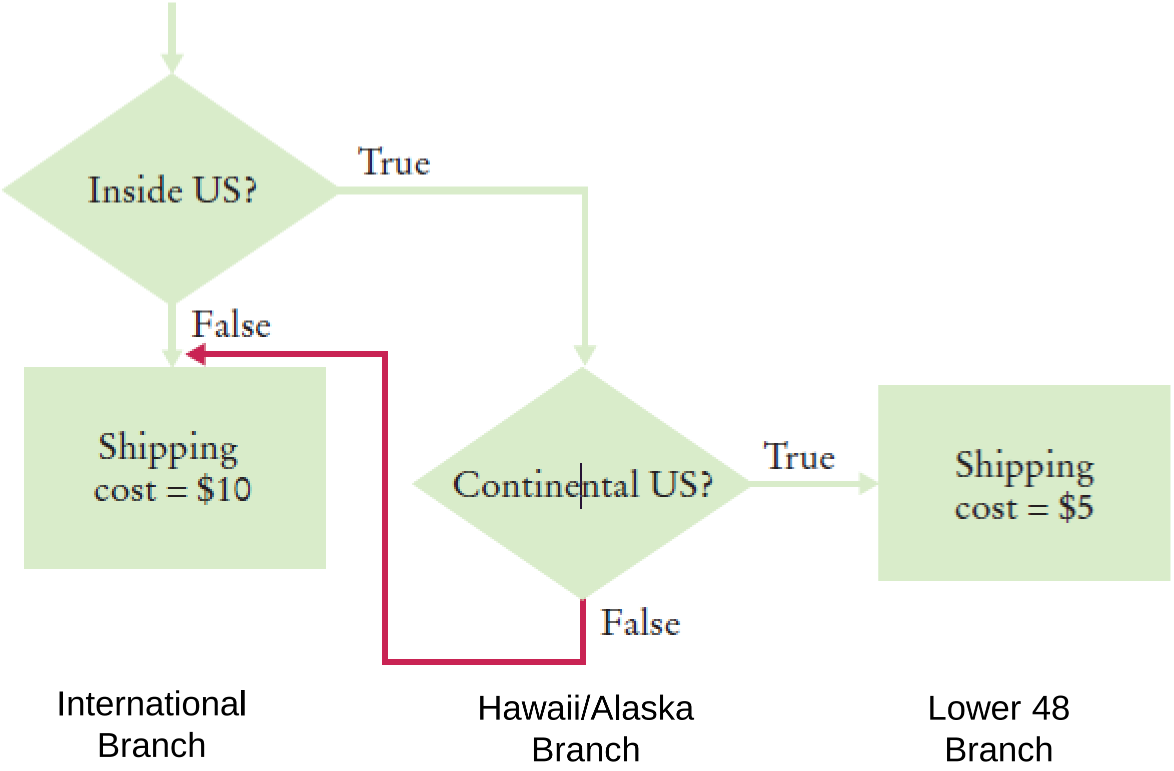
Hawaii/Alaska Branch

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.

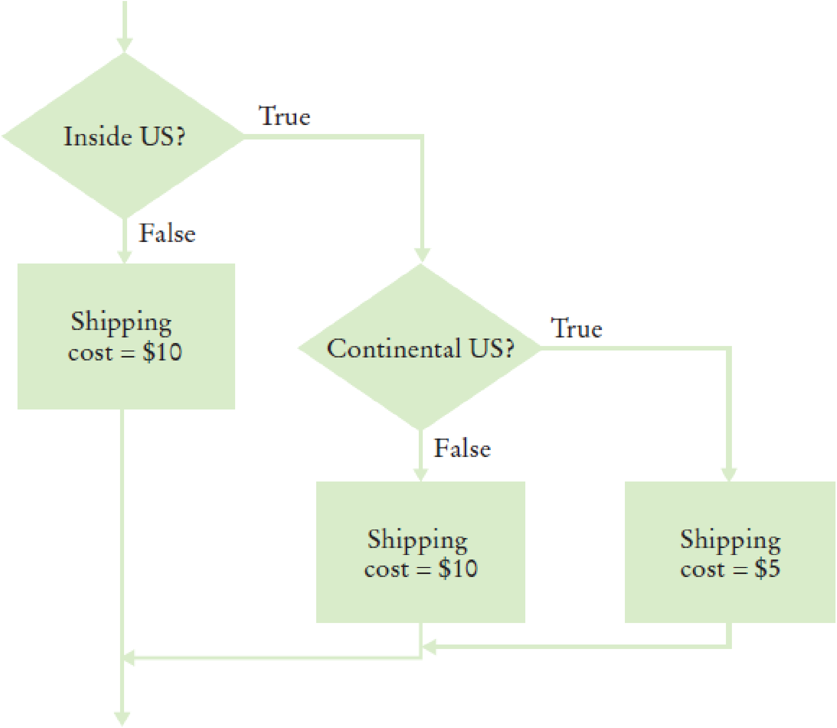
- **Do not do this!**



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



Problem Solving: Test Cases

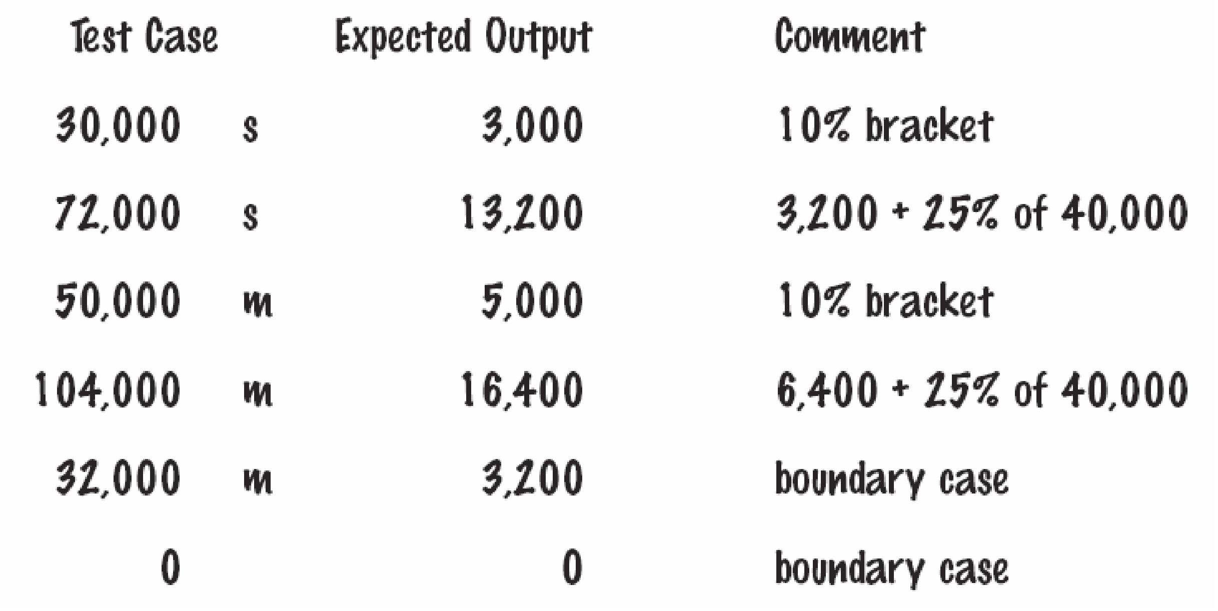
- Aim for complete *coverage* of all decision points:
  - There are two possibilities for the marital status and two tax brackets for each status, yielding four test cases

  - 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

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))
{
   ...
}
```

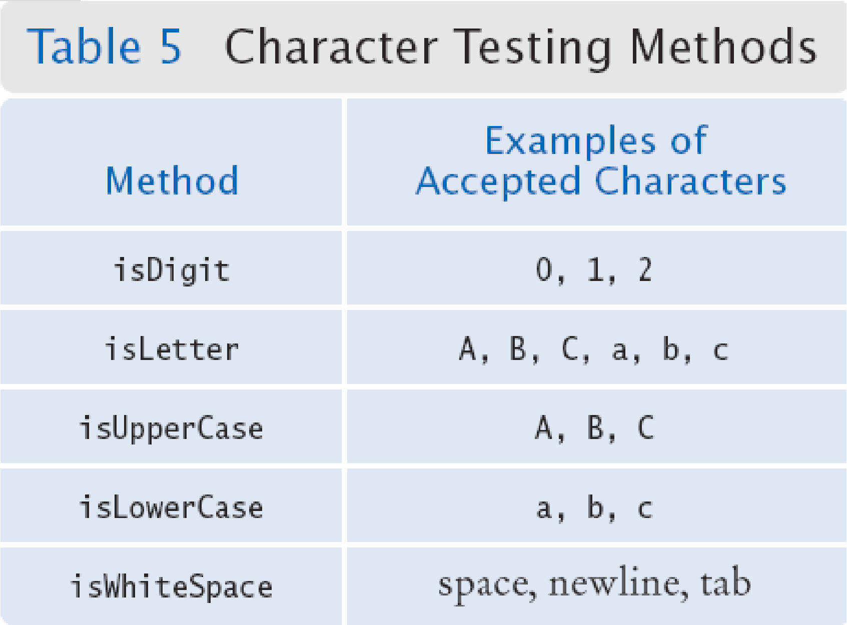


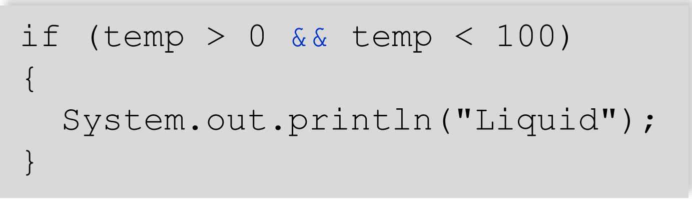
Table 5 Character Testing Methods

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

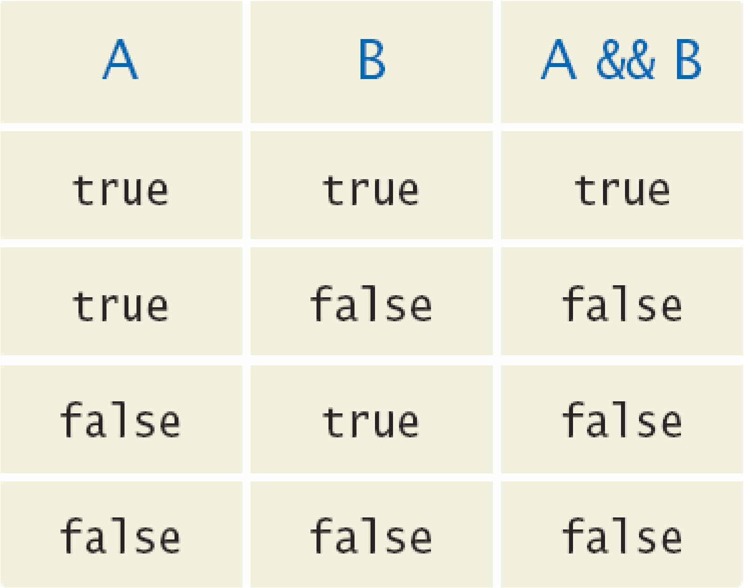
Combined Conditions: &&

- 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");
}
```



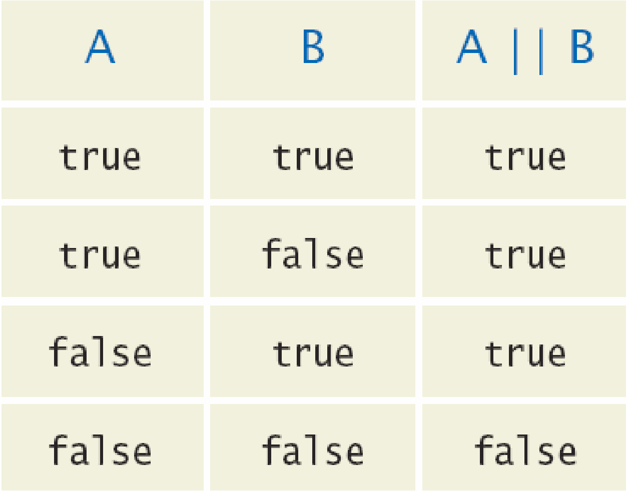
A	B	A && B
true	true	true
true	false	false
false	true	false
false	false	false

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



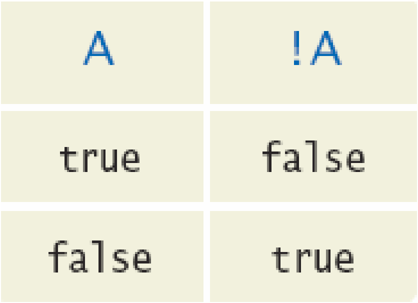
A	B	A || B
true	true	true
true	false	true
false	true	true
false	false	false

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

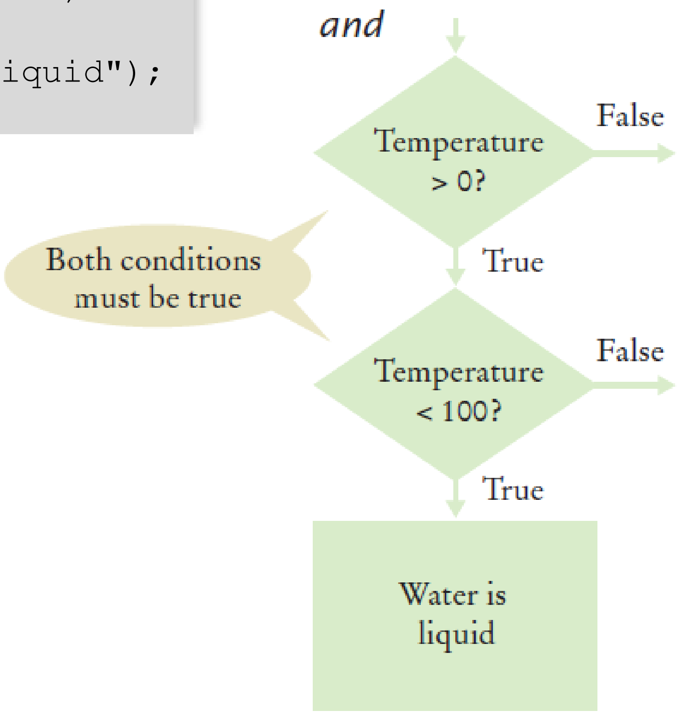
- If using !, try to use simpler logic:

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

and Flowchart

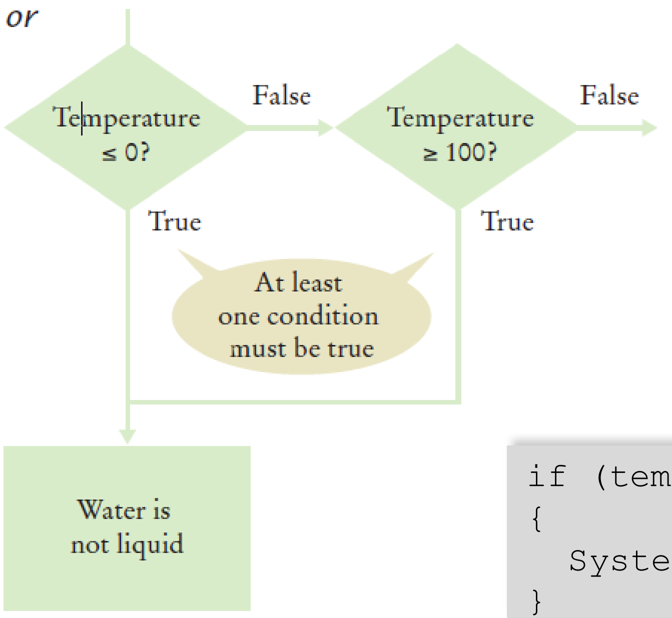
- This is often called ‘range checking’
    - Used to validate that input is between two values

```
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

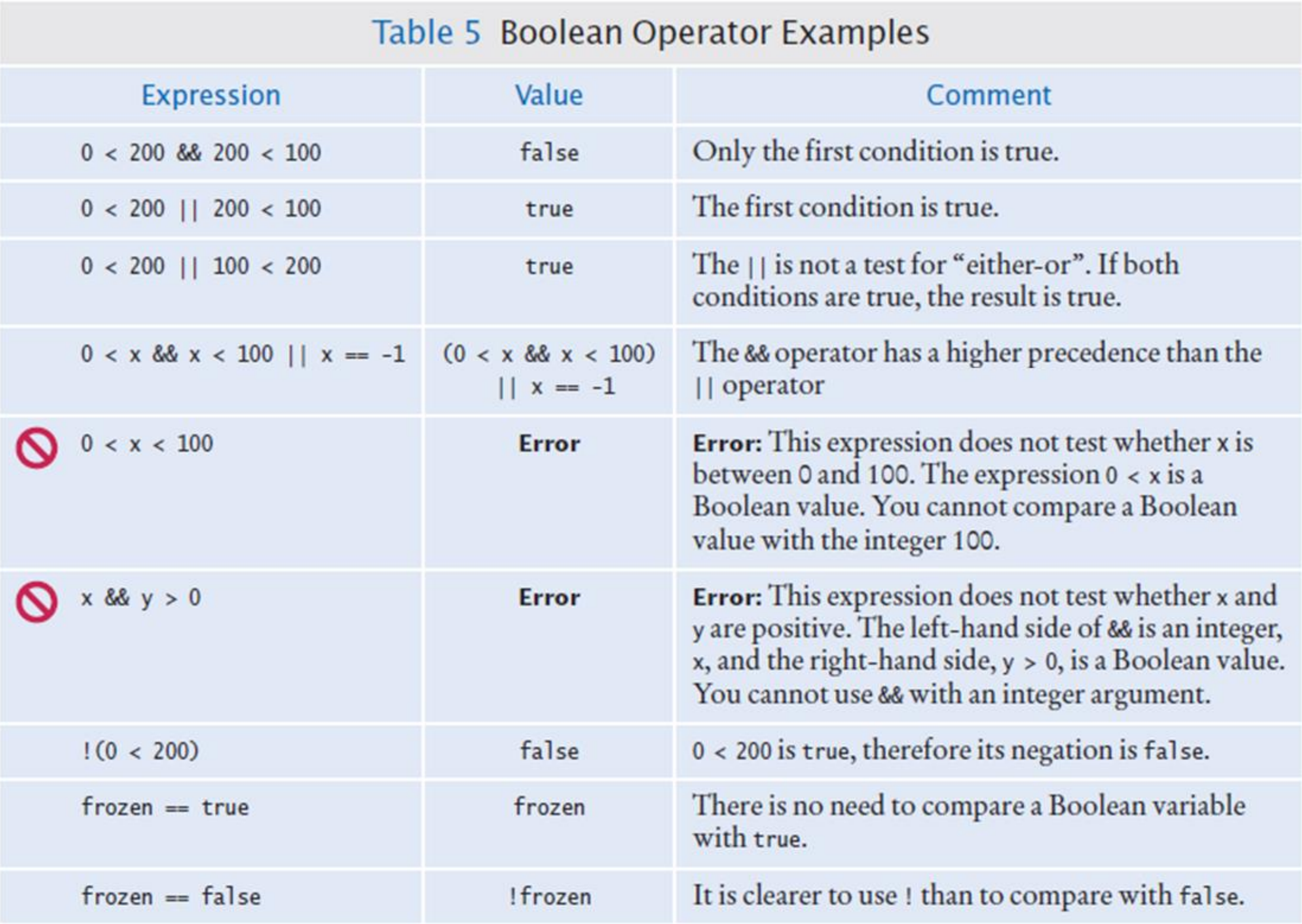


Table 5 Boolean Operator Examples

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
🚫 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

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

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

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

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

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

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

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)
```

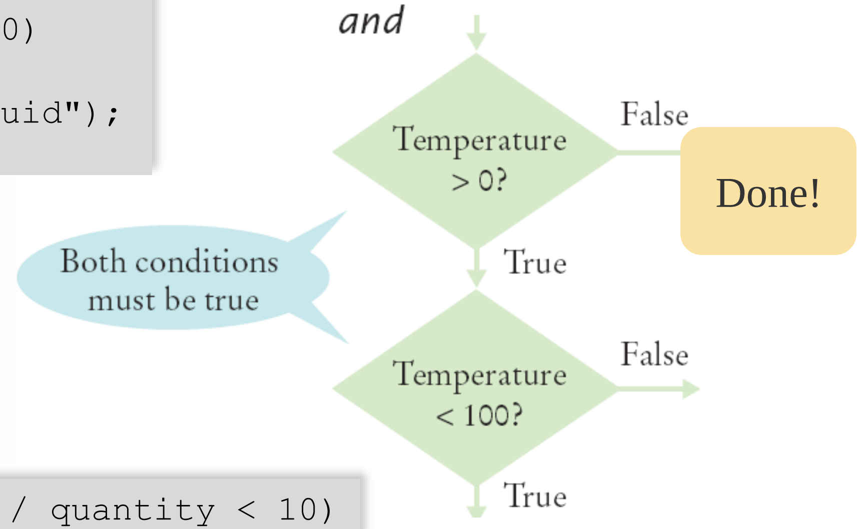
Common Error

- 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");
}
```

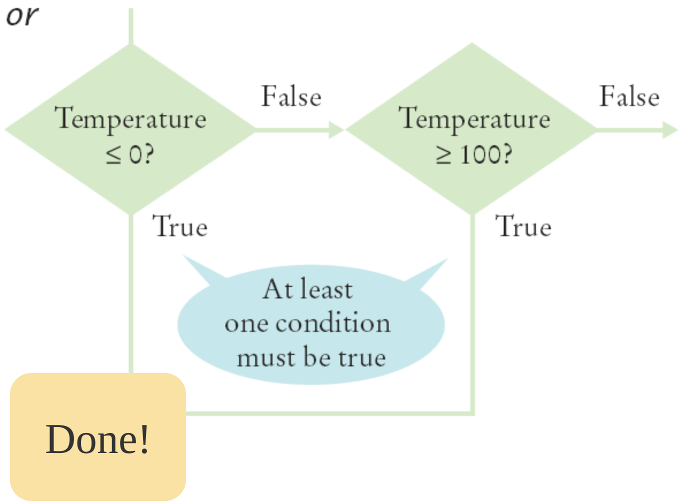


```
if (quantity > 0 && price / quantity < 10)
```

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");
}
```



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.